AN INFORMATION VISUALIZATION SOLUTION FOR THE ANALYSIS OF THE AFM SIMULATION OUTPUT DATA

THESIS

Stuart H. Kurkowski, Captain, USAF

AFIT/GCS/ENG/00M-11

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/000M-11

AN INFORMATION VISUALIZATION SOLUTION

FOR THE ANALYSIS OF THE

AFM SIMULATION OUTPUT DATA

THESIS

Stuart H. Kurkowski, M. S.
Captain, USAF

AFIT/GCS/ENG/00M-11

AFIT/GCS/ENG/00M-11

AN INFORMATION VISUALIZATION SOLUTION

FOR THE ANALYSIS OF THE

AFM SIMULATION OUTPUT DATA

THESIS

Presented to the faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science (Computer Science)

Stuart H. Kurkowski, M. S.

Captain, USAF

March 2000

AFIT/GCS/ENG/00M-11

AN INFORMATION VISUALIZATION SOLUTION

FOR THE ANALYSIS OF THE

AFM SIMULATION OUTPUT DATA

Stuart H. Kurkowski, M.S.
Captain, USAF

Approved:

[illegible] 29 Feb 00

Lt. Col. Timothy Jacobs (Chairman) **date**

Thomas C. Hartrum 29 Feb 00

Dr. Thomas Hartrum **date**

[illegible] 29 Feb 2000

Maj. Michael Talbert **date**

Acknowledgments

This visualization research was only possible with the help and support of many different folks. I would like to thank God for allowing me to be here and have the health and dedication to complete this monumental task. My family, especially my wife and daughter were the difference in getting me through it with the right approach and attitude. Thanks also to my Father, Wife and friend Tim Jeannette for help proofread this document. I would like to thank my advisor Lt Col Tim Jacobs, who put his trust in me as a student to complete this research for Air Mobility Command (AMC). His assistance and instruction provided me all of the skills I needed to run with the research. I also want to thank Capt Lee Maynard who helped me with SC support. I would like to thank my fellow students who kept me sane and focused on the task at hand. I would also like to thank Mr. Dave Doak the system administrator for the lab used to complete the research. Without Dave's support in configuring machines and getting software going, I would not have been able to complete this research. I want to especially thank the AMC support of Capt Tim Smetek and Mr. Gene Miller. Their effort in getting me up to speed on the Airlift Flow Model got everything started. I would also like to thank the members of my committee, Dr. Thomas Hartrum and Major Mike Talbert.

Table of Contents

List of Tables

Table of Figures

Table of Color Plates

AFIT/GCS/ENG/00M-11

Abstract

With the advancement of computer hardware and software computer simulations are now able to run faster and track more elements than ever before burdening the analyst with more and more data to analyze. Air Mobility Command's (AMC) Airlift Flow Model (AFM) is the Air Force's logistics simulator that simulates multi-day mobility scenarios in a matter of minutes producing megabytes of output data. Because the analysts' needs for summaries, trends, and comparisons of the data have surpassed the capabilities of current desktop spreadsheet analysis techniques new tools are needed.

This thesis looks at developing a robust information visualization architecture that integrates data processing, visualization, and user interaction, and supports reuse and component-based functions. This research develops a component-based 3-D visualization system for the AFM data. A domain-independent application framework is developed to support the component-based system design. This research also develops data reading objects, integrated data structures, and visual components as well as drill-down and user-interface components to produce an end-to-end visualization application for several aspects of the AFM data.

The results of this research show that an application framework can support information visualization applications. The use of a stable underlying framework architecture provides high levels of design and code reuse for future component development. The component-based functionality frees future development to concentrate on visualizing data and not the systemic concerns handled by the framework. This enables AFM and others to get a better return on investment for future work. The representative applications completed in this research already provide AMC with unprecedented insight into the AFM data.

AN INFORMATION VISUALIZATION SOLUTION

FOR THE ANALYSIS OF THE

MASS SIMULATION OUTPUT DATA

1 *Introduction*

1.1 Overview

With the continued increase in computer capability, computer simulations have proven themselves an effective tool of operational analysis. Simulations test systems that in real conditions are cost prohibitive or too dangerous to test. Simulations can evaluate operational plans and contingencies, highlighting problems or issues before the real operation. Simulations also provide a means for supplementing real world activities at a reduced cost. For example, the use of flight simulators supplements the training of a pilot in the real aircraft, reducing training costs for fuel and maintenance.

There are two types of simulations used to cover this wide range of needs. The first type are known as virtual simulations where real people operate simulated systems. In the second type, called constructive simulations, simulated people operate simulated systems [21]. Typically, constructive simulations use mathematical algorithms and other event manipulation techniques to transform the input data into results. An example of a constructive simulation is a stochastic, discrete event simulation used to simulate bombing a target. The simulation generates the bombs and their characteristics, as well as their effect on the target. The simulation uses mathematical algorithms to drop the bombs and calculate the damage to the target.

Running constructive simulations with the latest computer technology increases the speed of the simulations, the accuracy of the models, and the number of items the simulations can track. For example, most constructive simulations run faster than real-time producing results faster then ever before. Time acceleration is a benefit to the analyst who needs an answer now, not weeks from now. The speed can even enable an analyst to make several different runs to compare results. Increased data collection can also help the analyst look at aspects never before tracked.

Despite the advantages of new capability and constructive simulations there are some side affects of accelerating time and tracking so many objects. The most significant of these is the large quantities of data the simulations produce. The increase puts a burden on the user to process and analyze vast amounts of information. Often there is more data than current analysis methods can effectively process. For example, if constructive simulations can run a multi-day scenario in minutes, a real-world day's worth of data can be ready for analysis every few minutes instead of every 24 hours. If current analysis techniques take hours to conduct for each day's worth of data, the analyst will not keep up with the simulation's output. Additionally, at real-world speeds analysis can be done along the way to lead to a conclusion. In constructive simulations, the conclusion is available so quickly, along-the-way analysis is difficult and often overlooked. Analysts need some type of simulation "playback" for effective analysis.

Information is power, so getting information from the data is key to operational success. As John Peterson of Object/FX says in *The Visually Enabled Enterprise: Managing Information Through The Power of Visualization* [17], even though "data has been 'liberated', the fact remains that 98% of data is never looked at more than once". The reasons that Peterson gives for the lack of data use are as follows:

- The amount of data available is overwhelming and continues to grow.
- It is difficult to see integrated views of disparate data.
- It is hard to see patterns and relationships with current desktop tools.
- Users have trouble "drilling down" to the information they need.
- Data lacks context. It's not in a "meaningful" form that allows users to make decisions quickly and confidently.

Information visualization is one approach to help solve or at least reduce the problems associated with information overload. Visualization is not new; the scientific community has used it successfully for some time. Consequently, there are fairly good tool sets and documentation for scientific visualization. Information visualization, however, has less established standards and tools. Where scientific visualization is often developed for a single domain or algorithm, information visualization is much more broad and diverse. According to Judith Brown in her article on *The Euro-American Workshop on Visualization of Information and Data*, "Information visualization is different from statistical or scientific visualization in its effort to communicate the structure of information and improve access to large data repositories" [3]. Scientific visualizations typically have a single data set and form a static image or graph that can be directly analyzed. Most information visualizations are too large and complex for a single static image. As a result the complex images cannot be directly analyzed, so effective visualizations present the data in varying levels of detail.

This visualization research examines ways of using visual displays to support the decision-making process by making sense of the large amounts of data today's simulations can produce. This research compares the use of information visualization techniques on simulation output with the previously used graphing and spreadsheet-based analysis routines. Conclusions are drawn about the viability of using information visualization as a tool to support the analysis of simulation data. A constructive simulation for military logistics is used for this research.

1.2 Background

The Air Mobility Command's (AMC) Mobility Analysis Support System (MASS), and specifically the Airlift Flow Model (AFM) portion of the system, is a simulation system that faithfully models AMC's global air transportation system. It simulates policies, procedures, operations, aircraft, airbases, cargo, passengers, airfield resources, and other aspects of the air transportation system [6]. The simulation is based on numerous input files that are derived from operational plans (OPLANS) describing the aircraft, aircrews, cargo, air refueling, and numerous other aspects of a mobility operation. MASS produces megabytes (MB) of data in the form of more than twenty output files, summary files, and reports depicting different aspects of the scenario. The output includes aircraft launch times, aircraft cycle times, maintenance statistics, airfield statistics, crew statistics and more. The output generated from AFM is diverse both in type and context and can be very large. For example, the aircrew output is a single file comprised of ASCII characters, strings, and floats, that accumulates approximately 1.5 MB of data per scenario day. For a 180-day scenario this file alone can be 270 MB or more [6].

The output from a simulation may represent a single answer to some question, but often it's just a large amount of data. For the analyst to fully understand the results and their origin, they need additional processing and correlation. For example, a portion of the MASS AFM simulation produces output data for each day of the scenario. To track and compare a single element of data for a 180-day scenario, which is typical for MASS, the analyst must correlate that value in each of the 180 instances of the data. Other portions of the output are recorded in one-hour increments, which means tens-of-thousands of entries for a 180-day scenario.

The desktop analysis tools used today by the AMC analysts only provide spreadsheet type functions and mathematical charting. These tools can do sums, averages, and similar operations on a single set of data. These tools can also plot the sums and averages as line graphs or pie charts. These tools do not provide the needed capabilities such as correlation, integrated views, relationships, context, and drill down discussed by Peterson [17]. According to Judith Brown, the main theme behind information visualization is "to merge user interface, scientific visualization, and database disciplines to aid the decision-making process" [3]. Visualization systems can do the charting and mathematics like the desktop tools, but they can also provide these additional functions like correlation, context, and user interaction.

Context is defined as the surrounding or supporting information used to present the information to the analyst or user. The context can be as simple as using dollar symbols when showing currency or as elaborate as plotting geographical based data on a map. The context enables direct correlation and comparison of entities by the human visual system. The visualization can also provide the context of time by enabling the user to step through the days of the simulation, providing that much needed “playback” capability. By providing a context for the visualization and providing the user a means to control the representation, the user's cognitive processes spend more energy on decision-making.

The user interaction can range from user navigation through a scene to user control over what data is shown and when. The user interface can also enable the user to change the representation or control how much information is presented. A good visualization system will enable the user to view micro and macro levels of data, providing varying degrees of detail. For information visualization systems, this context and user interaction is critical and provides the biggest divergence from scientific visualization and current tools.

Due to the broad audience and popularity of information visualization systems, they need to have architectures or frameworks that make the mixing and matching of data processing and application components straightforward. Scientific visualization systems are often monolithic and centered on a particular domain or data type, so the dynamics are not as critical. Information visualization systems must support various contexts, diverse data sets, and flexible interfaces. Even for a single simulation such as AFM, the diversity of the data forces the use of different visualization techniques in the same system. Therefore, the architecture must be able to support the mixing and matching of these techniques as they are developed.

Having a broad audience means there is no guarantee the user of the information visualization system will be a scientist intimately familiar with the domain. The interface designs must support a variety of users. Supporting these diverse components and interfaces is what makes information visualization architectures robust. An architecture that is not robust forces developers to build new systems each time something changes.

1.3 Problem Statement

Develop a robust information visualization architecture that integrates data processing, visualization, user interaction, component-based functions, and supports reuse. Demonstrate the capabilities of the architecture by implementing component-based visualizations of the MASS AFM output. The applications must maintain the detail and accuracy of the simulation output data and must present this data in a manner that facilitates comprehension and exploration at all levels. This architecture must enable future information visualization researchers to concentrate on the visual aspects of the data and not on the underlying framework engine.

1.4 Research Objectives

The overall goal of the research is to show that information visualization improves analysis of simulation data. This improvement must include development as well as the use of new capability. If the visualizations are an improvement, but the only way to add the capability to the system is to rewrite the system, the development overhead will outweigh the visualization benefit. For this reason the first objective is to develop a robust information visualization architecture that supports component-based system development. The second objective is to develop specific application components for this framework that support reuse for future development. The third objective is to implement applications that visualize the AFM output to improve analysis of the data. This visualization research uses these applications of the AFM output to show improvements in analysis and to validate the architecture.

1.5 Scope and Limitations

This AFM visualization research implements several visualization applications that represent a portion of the MASS output data. These applications are intended to determine the value of information visualization and demonstrate the dynamics of the architecture. These applications are not intended to be a visualization of the entire set of MASS output data.

The data visualized in this AFM visualization research is actual unclassified MASS output data as produced by the system. No attempt has been made to correct or derive data beyond what is produced by the simulation execution.

1.6 Methodology

Several information visualization research/prototype systems exist today; however, none cover the requirements of the MASS visualization completely. These research efforts have positive aspects ranging from data structure usage to visualization techniques, but all of them lack the key element for this AFM visualization, which is showing relationships and drill down of the data. Therefore, the positive aspects of these current efforts were synthesized in the development of this visualization research.

Based on the ideas of these current research efforts, the AFM information visualization system can be broken into three main areas 1) data processing (retrieval and population of the data structures), 2) graphics rendering, and 3) user interaction. This AFM research addresses these in data flow order, starting with data processing and proceeding through to user interface. The individual aspects of these three areas are further broken down into framework "modules" and application "components". This visualization research uses an iterative two step process. The first step is the framework development. The framework is the consistent visualization engine that is implemented in the form of modules for each of the three areas. The second step is the development of the application components. The application components plug into the system framework modules using defined interfaces. This approach puts emphasis on the development of a complete framework and further reinforces the idea of a component-based architecture by using it from the onset of development.

For the first step of the development, the existing research ideas were reviewed and incorporated into an architectural framework design. This design includes module and interface definitions. The module and interface designs were then implemented in the form of framework modules and abstract base classes interfaces for the components of the second step.

For the component level pieces, the second step requires the development of the data retrieval, storage, and indexing of the AFM output data as well as the contextual map data. It also requires the representation and user interaction for the data. The process for developing and implementing this second step is listed here.

1. Select portions of the AFM output data set that will provide coverage for the framework success criteria.
2. Produce AFM output files from a valid execution of the system. Read and parse the selected output files into appropriate data structures.
3. Develop visual representations for the AFM data. This includes both the micro and macro level of detail as well as the drill-down displays.
4. Develop the user interface appropriate for the exploration of the data and context.
5. Validate the component's functionality with the other portions of the architecture.

The theme of the objectives of this AFM visualization research is to demonstrate the benefits of this architecture through reuse and component independence. This is accomplished by producing multiple visual applications for different aspects of the AFM output data. These different applications are used for metrics, comparison, and validation of the architecture. To measure the success of the AFM visualization research goals, the following metrics are used.

1. One measure for the first objective is testing the capacity of the system to include or exclude components. Different combinations of the components are tested to make sure the framework and system still operate without problems.

2. Another test of the first objective's criteria is evaluating the number of component classes that include other components directly. If the components are independent they communicate only through the documented interfaces of the architecture.

3. For the second objective of code reuse, the reuse metric tracks the number of lines of code a new component uses from a previously completed component. This value is then compared with the total lines of code for the new component, to achieve a percentage of reuse.

4. For the third objective of visualization development, a subjective analysis is done to estimate the gain and ease of analysis with the visualization. This will determine if the visualization is more beneficial than current analysis techniques.

1.7 Document Overview

The first chapter describes the overall environment in which this thesis was developed and the role it plays in the analysis of MASS output data. Chapter 1 also defines the objectives of the research. Chapter 2 discusses the various visualization architectures and related research efforts being used in information visualization today, and then discusses visualization tools and techniques used in management and analysis visualizations. Chapter 2 also contains a detailed discussion of the MASS simulation. Chapter 3 contains the software design and methodologies used to achieve the first two objectives and the development process undertaken by this AFM visualization research. Chapter 4 discusses the two implementations completed for this AFM visualization research using the system framework. Chapter 5 summarizes the results of this visualization research and its impact on the MASS analysis. Chapter 6 discusses conclusions and recommendations for future research.

2 *Background*

The first part of this chapter looks at the architectural issues associated with the dynamic data types and visual representations in an information visualization system. The second section highlights architectures and data representation approaches of other research efforts and tools. The third section summarizes some of the current information visualization techniques. The final section discusses the details of the MASS AFM simulation.

2.1 Software Architectures

Software architecture defines the style used to organize a software system. This style helps to structure the flow of control throughout the system and defines which portions of the system handle each of the required tasks and computations [11]. An architecture also establishes standard techniques used for communicating and accessing data in the system [9]. By defining the architecture, designers and developers can have a better understanding of the system's operation, making development of additional system components easier.

To set a basis for discussion of architectures Garlan and Shaw [9] use several key terms. The first is the idea that an architecture is broken down into pieces they call *components*. The communication between these components takes place through the architecture's system of *connectors*. Additionally, the behavior of the system must adhere to certain *constraints*, which are rules for combining the components and connectors. The constraints help decrease the system's complexity and improve basic understanding. Garlan and Shaw use these terms to discuss various architectural styles.

The first of these styles is the pipe and filter style. In this style a component reads streams of data as input and produces streams of data as output [20]. These input/output connectors represent the “pipes” of the style. The processing or transforming of the data by the component itself is the “filter” portion of the style. By its nature the pipe and filter architecture is a data flow type system. An advantage of the pipe and filter style is that viewing the system as a composition of filters makes the system easier to understand. Another advantage is that the filters can be reused in other systems as well as a different order in the same system. The only requirement is that consecutive filters agree on input and output formats. A disadvantage of the pipe and filter style is they usually are not good for interactive systems. It is difficult to interact with intermediate points along the flow, because a single data set flows from end-to-end. Another disadvantage is that pipe and filter styles are difficult to develop and often end up forming a batch sequencing system [20].

Batch sequencing is a style similar to pipe and filter in its structure, but it differs in execution, because each filter processes all of its input data before passing it on as output. Batch sequencing can be good or bad depending on the requirements. If the requirements for using a pipe and filter are to get the data flowing through the system, then a batch sequencing implementation is bad and will result in filters being idle. However, if batch sequencing parses up the data into meaningful pieces for faster processing, batch sequencing can be a benefit.

Another style is the data abstraction/object-oriented architecture. This style is based on the idea of a component being an object that maintains the state and integrity of the resource. The component’s methods are the connectors for this architectural style. The advantages of this style are the object-oriented benefits of encapsulation and information hiding. Additionally, a self-contained object lends itself to reuse. A disadvantage of the style is that for one object to communicate with another object, it must know the identity and interface of the other object [20]. Having to know the identity of the other object and include it in the compilation of an object can limit reuse and have rippling affects if changes are made to a shared object.

The fourth style is the layered systems style. "A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below" [20]. Components in this style are the objects or structures that make up a layer. The connectors are the interfaces defined for each layer and its client or server. Some typical constraints are limiting interaction to adjacent layers. The classic example of a layered style is the International Organization for Standards (ISO) Open Systems Interconnection (OSI). The ISO OSI network protocol stack ranges from the lowest data link and physical layers to the highest application layer. One advantage of a layered style is that different implementations of a layer can exist, as long as it maintains the interface to the other layers. A disadvantage is that most systems cannot be divided up into layers. For example, performance requirements may even dictate combining layers, or coupling two non-adjacent layers so tightly that middle layers are ignored. This violates the principles of a pure layered style, making it hard to layer this particular example.

The fifth style is the repository type architecture. "In a repository style there are two quite distinct kinds of components: a central data structure represents the current state and a collection of independent components operate on the central data structure" [20]. The interfaces in the style vary depending on the implementation. Some variations of this style provide means for operation-based components to directly access the data and execute on the data. Components operating at will against the data store make the data store behave like a database. Another variation is called "blackboard" where components operate based on triggers from the current state of the data store. Components have direct access, but operations are based on the current condition of the data store not the individual components.

Garlan and Shaw conclude with the realization that most large systems are designed by combining more than one style. This heterogeneous style can take advantage of the positives of a particular style and combine it with another style. This provides a means for the designer to use the positive aspects of one style to cover the negative aspects of another style [9].

2.2 Visualization Tools

As mentioned in Chapter 1 of this paper, several visualization tools have been developed as a result of the demand for information visualization. These range from scientific visualization and medical visualization to general data analysis tools. This section discusses three of these relevant to this AFM visualization research. None of the three tools meet the needs of the MASS visualization in its entirety, but each one has many aspects that contribute to the final outcome of this research. The first one is *The Visualization Toolkit,* which highlights the use of a pipe and filter architecture and the extensive use of array data structures. The second tool is *Visage,* which utilizes an object-based repository style architecture and multi-dimensional data visualization. The third is a research prototype called *Swift-3D,* which has a batch sequence architecture, designed to handle large data sets. The *Swift-3D* research also provides the pattern for the basic module layout for this research.

2.2.1 Visualization Toolkit

The Visualization Toolkit (*vtk*) is a collection of text and software that "describe visualization algorithms and architectures in detail and provides a working architecture and software design for application of data visualization to real-world problems" [19]. The *vtk* software provides a variety of data readers with the capability to read straight ASCII, bitmaps, texture maps, and even binary data sets. The architecture used by *vtk* is true pipe and filter. Figure 1 shows how *vtk* implements the pipe and filter architecture.

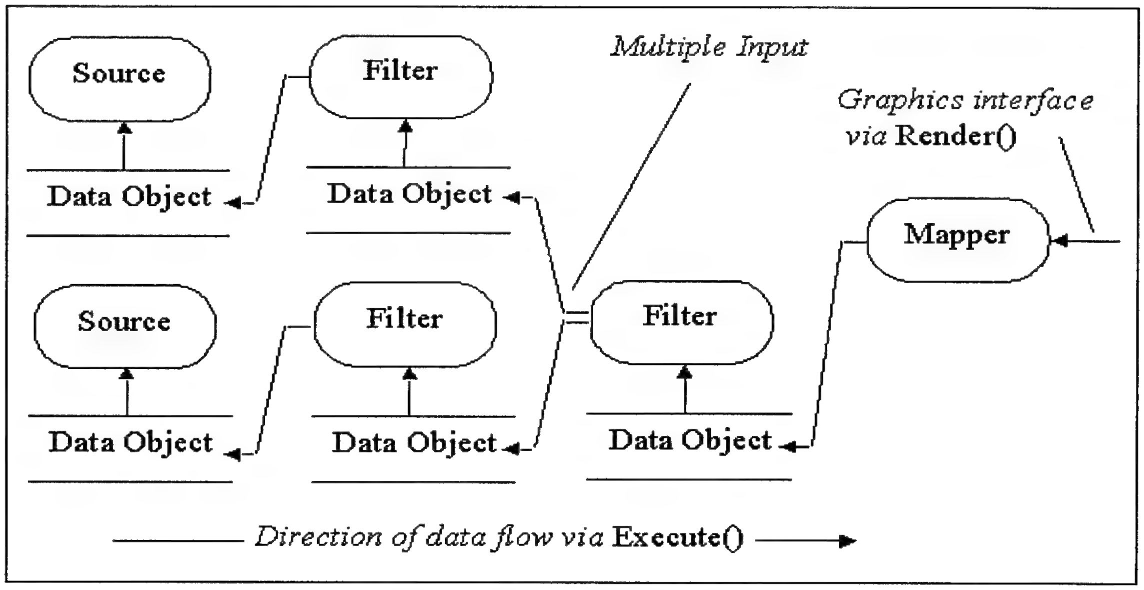


Figure 1: The VTK Visualization Pipeline

An application developed using *vtk* is defined by its unique combination of *vtk* readers and filters. The data in a *vtk* based application flows from the source at the left in Figure 1 through the filters to the Mapper and Render classes for display. The readers are used at the left end of Figure 1 to read the source files and populate the data structures. Once the data structures in the data objects are populated they are passed to the right through filters before they are rendered on the screen. There are numerous filters ranging from ones which draw an outline around the outermost points, to ones which apply a glyph to each data point. There are over 95 filters available today and custom filters can be added [19]. The output from the last filter is sent to a mapper object that translates the points into graphical primitives for rendering on the screen.

The execution arrows, shown in Figure 1 between the objects, may seem to be going in the wrong direction, since they run counter to the data flow, but they are actually correct. As mentioned in the architecture discussion, pipe and filters can often be implemented incorrectly as batch sequence styles. *Vtk* ensures the true flow of data by utilizing an `Execute()` method for each object. In *vtk*, the output data of an object is owned by that object. When an object gets some input data, the data, not the object knows where it came from. This keeps the filters independent of each other, but still enables *vtk* to run the pipeline on demand from the right end of the pipe. For example in Figure 1, the `Render()` method renders the image, to get its data, the Render object calls the `Execute()` method of the owner of its input data (Mapper). The Mapper in turn calls the `Execute()` method of the owner of its input data. This process continues to the left until the chain of requests gets to a source object. When the source object is reached, data is read in and the data flows back through the pipe. Each object processes the data in its `Execute()` method and then passes it on to the requester. The data finally ends up at the `Render()` method where it is displayed for the user [19].

There are two limiting factors that prevent *vtk* from being a complete solution for the MASS visualization. The first of these limitations is that *vtk* system, only handles 3-D Cartesian coordinate based data sets. Input format is constrained to three columns representing the x, y, and z coordinates followed by scalar data values depicting a single attribute for each point. For datasets like temperatures, wind velocities, or particle counts this is possible, but in AFM, the data is too diverse and abstract for this format.

The second limitation is that *vtk* is geared toward the visualization of a single data set, with analysis of that data being done by the use of a combination of filters. To manipulate the display of the data in *vtk* requires the application developer to change the combination of filters compiled into the executable. This limits the visualization to a single static image of the data. There is no drill down capability or any way to directly relate two datasets. This also limits the user interface to basic rotation of the object and zooming in and out.

The use of an "execute" method in each object minimizes memory usage by only running those modules that are needed. It also keeps unused data from ever getting read in, saving on memory and input/output time during execution. The AFM visualization research incorporates *vtk*'s optimization of the execution path and pipe and filter architectural design to improve its efficiency. The AFM data sets are large, so optimization of their storage and execution is important to this research.

2.2.2 Visage

"*Visage* is a prototype user interface environment for exploring and analyzing information. It represents an approach to coordinating visualizations and analytical tools in data-intensive domains" [15]. *Visage,* like *vtk,* is based on a single data set, but it provides a means for the user to determine the display of the data, not the programmer. Figure 2 gives an example picture of the dynamic display that the user has in *Visage*. *Visage* gives the user an interface to

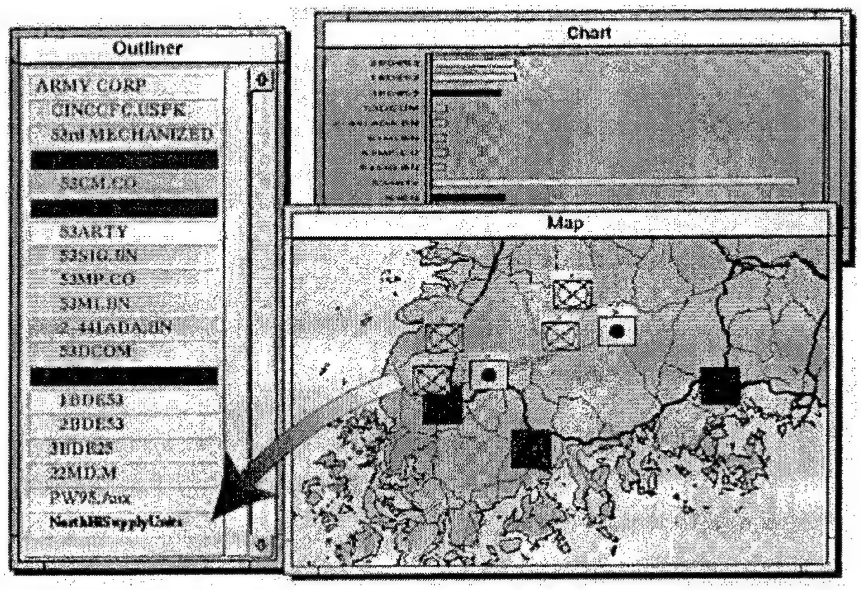


Figure 2: Sample Visage User Interface

display the data in numerous different display formats that are defined by frames. These frames range from spreadsheets, to bar and pie charts, to geographic renderings on a map context. For example, Figure 2 shows the representation of logistics data using map plots, bar charts and spreadsheets. The user controls the display by opening a blank frame and then dragging a visualization element into the display. For example, if the user wanted to see the logistics data in a bar chart, they open a new chart frame and drag the unit from the map to the chart frame.

The system's use of data objects called visualization elements is the key to its ability to drag and drop the same data into different views. Each visualization element is present in the system as only one object in the repository. This enables the frames to stay consistent and accurate. For example, if the user changes the color of some element, it changes color on all of the frames that are currently displaying it [14]. Furthermore, each attribute of the element is set to display or not display based on the type of frame. When an object is dragged onto a frame, that frame object renders those attributes flagged for display in that type of frame. For example, in Figure 2, when the user has the logistics data in the outliner (the Visage spreadsheet interface) the object's "outliner" attributes are displayed in spreadsheet form. When the unit is dragged to a bar chart frame, the supply quantity attributes are plotted. In the map frame, the latitude/longitude attributes and the unit's symbol attribute are displayed. The user can change the attributes flagged for each frame with pull-down menus [15].

Visage is limited in that it is only a prototype and relationships between data are missing. *Visage* provides the user many different ways to look at the same data set, which aids in doing good comparisons and analysis of like data, but the user is limited to the current data set. It is difficult for the user to drill down on a piece of data and see what is behind it or see how it relates to another data set. Additionally, *Visage* uses databases as its data input and database queries to populate the visualization elements and their attributes. The AFM data would have to be modified to fit into this format.

The use of data objects to support broader data types and visualize that data in many different ways are the main contributions of *Visage* to this AFM visualization research. The variety of visualizations helps satisfy some of Peterson's [17] issues with data context and integrated views. However, like *vtk, Visage* is still lacking the ability to see patterns and relationships, and drill down in the data.

2.2.3 Swift-3D

Swift-3D is a prototype visualization system being developed by AT&T Laboratories for the visualization of geographical telecommunications data. The system combines a data collector module, aggregator module, and visualization interface module to build a visualization application [12]. *Swift-3D* was developed for visualizing very large data sets with tens of gigabytes of data. To handle these large data sets *Swift-3D* uses a batch sequence architecture. *Swift-3D* also has a specialized data file format and query language to minimize the database management overhead. The data collector assembles the results of the query in a unique self-describing data-independent binary format. This binary format contains the sequence of records and a header that defines the record size, type, and data context [12]. The data set is then passed on to the aggregator for visualizing the data. Knowing that the data sets are going to be large, "the visualization module explicitly controls paging via memory-mapped files" [12].

Swift-3D is an example of a system that uses batch sequencing to its advantage. The large data sets are divided up into manageable pieces that can be quickly processed by the three system modules. For example, an hour's worth of telecommunications data can be divided up into small enough pieces that the system can render them in seconds. The batch sequencing enables the analyst to see the data faster than real-time, making simulation "playback" possible. Using menu selections the user can force different batches to be created on demand.

The three-module architecture of *Swift-3D* is the basis for the design of this AFM research, however the proprietary areas of the AT&T research effort make it impractical as a complete solution. By combining the idea of dividing the data into manageable pieces and utilizing the *vtk* "execute" method to process those pieces across the three system modules from *Swift-3D*, the AFM visualization research has a good basis for an architectural design.

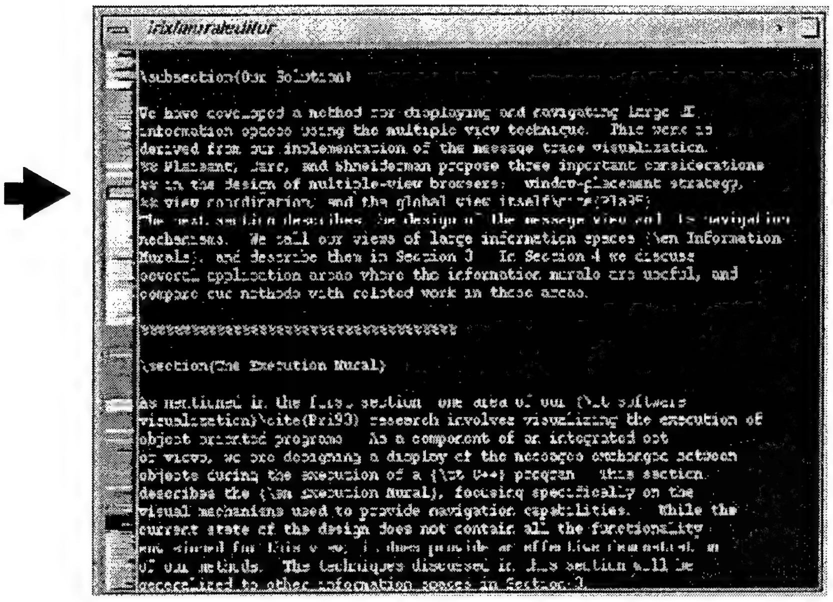
2.3 Visualization Techniques

Because data in the world of information visualization is often abstract, data structures exist to hold the data, but visual objects don't always directly represent the data. Additionally, the data sets can be too large to display completely on a single screen. This section examines several different approaches to data visualization that attempt to address these issues.

2.3.1 Information Murals

Information Murals, introduced by Jerding in *The Information Mural: Increasing Information Bandwidth*, are a visualization concept used to increase the user's understanding of large data sets [10]. With information murals the user can examine details of the data within the context of the entire data set. According to Jerding, "being able to see some representation of the entire information space provides an initial gestalt overview and gives context to support browsing and search tasks." The information mural technique does not address the specific visualization of the data itself, but merely a way to give the user a sense of the information space. It is a positive step for helping users of information visualization systems get a handle on the magnitude of the data set. Especially if the data is abstract in nature, the context is a key element for the user's ability to understand and navigate the representation of the data.

Because the screen size of today's computers is limited, not everything can always be shown clearly to the user. To address this problem "the mural creates a miniature version of the information space using visual attributes such as grayscale shading, intensity, color, and pixel size, along with anti-aliasing compression techniques" [10]. Figure 3 shows these techniques in



```
\subsection{Our Solution}

We have developed a method for displaying and navigating large
information spaces using the multiple view technique.  This work is
derived from our implementation of the message trace visualization.
%% Plaisant, Carr, and Shneiderman propose three important considerations
%% in the design of multiple-view browsers:  window-placement strategy,
%% view coordination, and the global view itself\cite{Pla95}.
The next section describes the design of the message view and its navigation
mechanisms.  We call our views of large information spaces {\em Information
Murals}, and describe them in Section 3.  In Section 4 we discuss
several application areas where the information murals are useful, and
compare our methods with related work in these areas.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\section{The Execution Mural}

As mentioned in the first section, one area of our {\it software
visualization}\cite{Pri93} research involves visualizing the execution of
object oriented programs.  As a component of an integrated set
of views, we are designing a display of the messages exchanged between
objects during the execution of a {\tt C++} program.  This section
describes the {\em execution mural}, focusing specifically on the
visual mechanisms used to provide navigation capabilities.  While the
current state of the design does not contain all the functionality
[illegible]
of our methods.  The techniques discussed in this section will be
generalized to other information spaces in Section 3.
```

Figure 3: Information Mural of Large Document Text Editor

use with a large document text editor. The entire document is shown in the slider on the left side of the window, while the currently viewed area is highlighted with a box, pointed to by the arrow.

A limitation to the technique is that it compresses the original visualization based on the largest piece of the image. This means that if the original image has pieces relatively small compared to the largest piece, they may be hidden in the compression. For example, in the text editor of Figure 3 if the widest page of the document is a lot wider than another section, the smaller section could be very small when compressed proportional to the large section. Additionally, if the original visualization uses miniature representations of the data, the data points could very well be lost all together in the compression and anti-aliasing routines.

2.3.2 Information Glyphs

"Glyphs are graphical objects or symbols that represent data through visual parameters that are either spatial, retinal, or temporal" [5]. Glyphs can be as simple as bars in a chart, points on a scatter plot, or as abstract as the size of a circle on a map. In data visualization, glyphs are

]

intended "to expose patterns among sets of...artifacts and to help identify differences" [5]. The key to a glyph is encoding the data so comparisons can be drawn directly and easily.

Mei Chuah, in *Information Rich Glyphs for Software Management Data* [5], outlines three rules to design an effective glyph. Her rules suggest using 1) small multiples, 2) established visualizations, 3) and information rich glyphs. "Small multiple designs contain small numbers of representations arranged on a grid and the designs are all based on the same structure" [5]. By utilizing small multiples, the density of the data points per inch of screen can remain high, while still enabling the user to make comparisons. Likewise if the structures are all the same, the human visual system can make the comparisons with minimal cognitive effort. Established visualizations are important, so the data being conveyed can be compared and used by the user. If interpretation of the glyph is difficult for the user, then it will detract from the presentation.

Figure 4 and Figure 5 show some information rich glyph examples that highlight aspects of Chuah's material [5]. Both depictions are set up for small multiples and they are both information rich by the amount of data shown in a small space. The first management glyph (Figure 4) shows ten characteristics of a software development team in one concise glyph. The

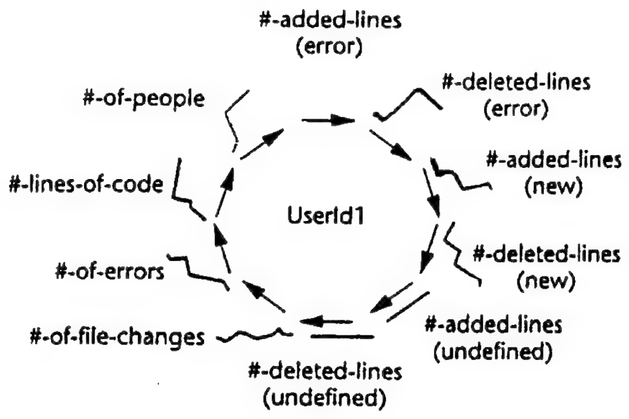


Figure 4: Management Gylph

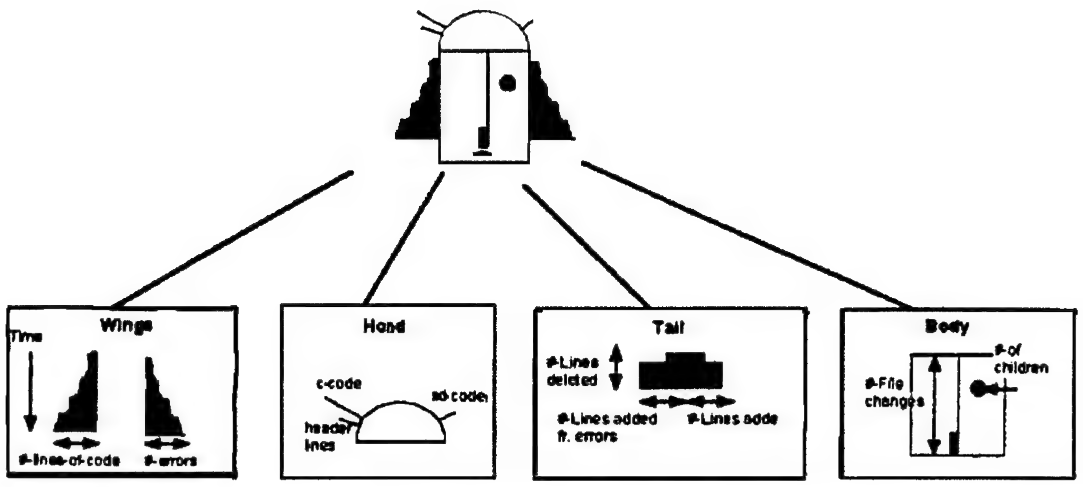


Figure 5: InfoBug Glyph

second glyph (Figure 5) shows over eleven characteristics of a software component in a single glyph. Figure 6 shows how grouping a set of these information rich glyphs together enables direct comparison and analysis. It is much easier to compare these characteristics across all sixteen

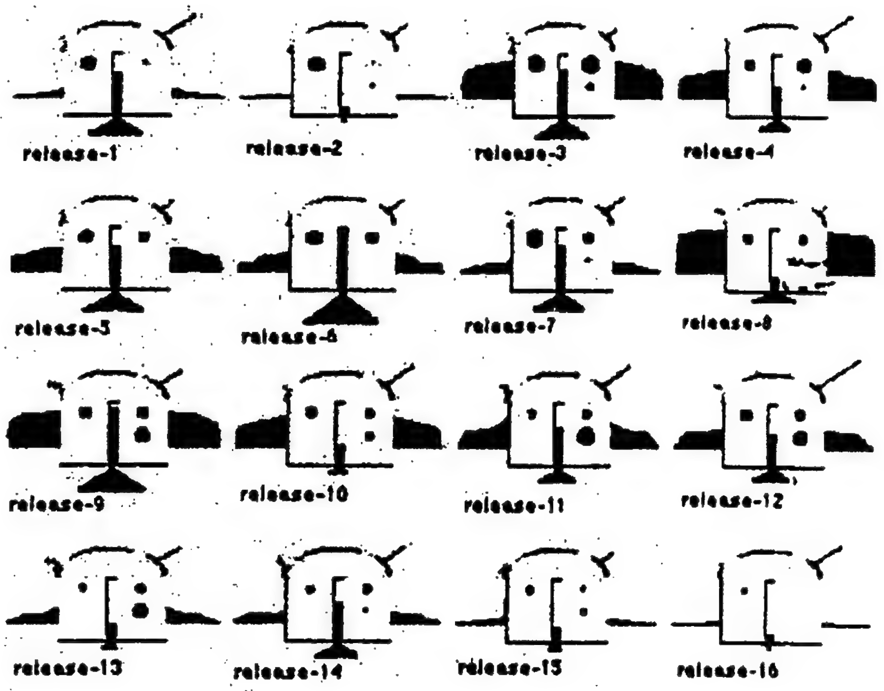


Figure 6: Example of Glyphs for Comparison

software releases using these glyphs than having to flip through individual charts on each component. The glyphs enable immediate and direct comparison with little effort.

2.4 MASS Airlift Flow Model

The Airlift Flow Model (AFM) is the major portion of the Mobility Analysis Support System (MASS). The AFM simulates the movement of AMC Logistics resources during an operation. The system is a stochastic constructive simulation based on flat file inputs. The input files are generated from operational planners as well as planning documents like Time-Phased Force Deployment Data (TPFDDs) and operation plans (OPLANS). AFM generates a variety of output files based on switches set by the analyst at the beginning of the execution sequence.

The format for the input and output files are defined in the *AFM Baseline Document* [1]. This format varies for each area of interest. For example, the location list file contains a four-character identifier for the International Civil Aviation Organization (ICAO), followed by a string of numbers representing the latitude, latitude minutes, longitude, longitude minutes, and initial constraints for four attributes. The leg information output file contains 20 different attributes per aircraft per stop per day. The data types in these files are diverse, ranging from character strings to integers, to floats. All of the output generated in the individual files is logged by each simulation day. The simulation also generates several summary and report files that attempt to consolidate the individual output files. The analyst must perform any final summations or accumulations, such as average, high, and low levels over the entire run.

The key to a simulation run is the `scenario` file, which is unique to each scenario. The `scenario` file contains the information about the switch settings, the input file paths, the output file paths, the date to start the simulation, and the number of days for the scenario. The `scenario` file is needed by the visualization system to locate the input and output files and parse them correctly. For instance, the output data based on an ICAO's location must get the ICAO's latitude and longitude from the location list input file.

2.5 Background Summary

The visualization of data is much greater than passing the data to the graphics engine. As seen by the background discussion in this chapter, the architecture, the data structures, the visualization representations, and the user interaction with the display play a major role in the process. The Swift-3D three-module design provides the starting point for the design of this AFM visualization research. The Visage research highlights the notion of many contextual displays and common data objects. The *vtk*'s use of the "execute" method to optimize the execution path is also a key contribution to this AFM visualization research. The chapter also highlights several ways to visualize the data itself within the system being designed, thus helping the user examine the data while not losing the surrounding context of the entire data set.

3 Methodology

This Chapter discusses the methodology and design used to address the goals of this AFM visualization research. It outlines the methods used to reach the design and implementation of this research, as well as describing the success criteria for the research. The design portion of this chapter starts with a description of the system's architecture and a system design overview. The overview is followed by a detailed discussion of the research's implementation.

All of the goals and objectives for this research contain reuse and decomposition. The first objective is to design a product that supports the decomposition of functionality into components. The second objective is to make the development of future functions largely an exercise in reuse of existing components. The third objective, a synthesis of the other two, is to develop actual components that improve on current analysis techniques for AFM. To meet these objectives the application must help the analysts view the data as well as help the programmer develop new functionality. An object-oriented approach was selected as the implementation method to best meet these AFM visualization research objectives.

Object-oriented programming has three distinguishing characteristics: data abstraction, polymorphism, and inheritance. Like an abstract data type, an abstract class represents an interface behind which implementation can change. Polymorphism is the ability for a single variable or procedure parameter to take on values of several types. Inheritance makes it easy to derive new objects from other objects [7]. These characteristics increase the potential for reuse of software design and code.

Although object-oriented programming techniques can increase the amount of reuse, this AFM visualization research needs more than this to satisfy its objectives. This research needs an application framework to organize and manage the objects in a useful application. This framework provides the basic modules and underlying structure of the AFM visualization architecture. According to Fayad in *Building Application Frameworks: Object-Oriented foundations of Framework Design:*

> A framework describes the architecture of an object-oriented system; the kinds of objects in it, and how they interact. It describes how a particular kind of program...is decomposed into objects. It is represented by a set of classes (usually abstract), one for each kind of object, but the interaction patterns are just as much a part of the framework as the classes. [7]

The benefit of using a framework is it "goes beyond code reuse, it provides reusable abstract algorithms and a high-level design that decomposes a large system into smaller components and describes the internal interfaces between components. These standard interfaces make it possible to mix and match components" [7]. Additionally, with defined interfaces, "new components that meet these interfaces will fit into the framework, so component designers also reuse the design of the framework" [7]. These added benefits are what this research needs to meet its objectives.

Another benefit of frameworks is what is termed "inversion of control" [7]. In traditional software reuse, the programmer reuses components from a library by calling them from a custom-written main program. The programmer decides the overall structure and flow of control of the application. However, according to Fayad, "in a framework, the main program is reused and the developer decides what is plugged into it. The developer's code is called by the framework code. The framework determines the overall structure and flow of control of the program" [7]. For a visualization application this inversion of control frees the programmer from systemic concerns and lets the programmer concentrate on the system's application components.

This research uses the object-oriented C++ programming language [13,16] and its Standard Template Library to implement the objects and framework. The user interface is implemented with the Fox C++ User Interface Library [8] and graphics are implemented with the OpenGL graphics library [22].

To fulfill the robust framework portion of the objectives, the system architecture provides a way to development independent reader, data, visual, drill-down, and user interface components. The specific areas of interest in the application framework that must be exercised and validated are listed in Table 1. For the first objective to be met, the third column of Table 1 must be satisfied through the sample applications. Additionally, the ability to develop new components of each type and link the components into the system measures the system's capability to add new functionality. A robust framework only has to be re-linked to include a new component.

Table 1: Implementation Success Criteria

Type/Name	**Description**	**Success Criteria**
Framework Interfaces	(Internal) Layer to layer communication of the framework	The framework can manage the applications and execution with the interfaces provided.
	(External) Component to framework communication	Application can compile and execute with components only using defined framework interfaces.
Data	Data structures used in Data Objects	Diverse data sets can be handled transparently by the framework and components
	Population of data structures	Data objects can be populated in whole or incrementally by the framework and applications
	Reading of data and preprocessing	Data can be read in directly or preprocessed transparently by the framework and application.
Visual	Micro and Macro views	System can dynamically provide macro and micro levels of detail when the user selects it.
	Context based displays	Complete data set viewing with a single context
	Pick items on the display	User can select display items for drill-down
	Animation or time based display	Display can handle changing of time of simulation
Plot	Single and Multi-tab books	Display of single and multi-tab pop-up windows
	Pick items on pop-up windows	User can select display items for drill-down
	Diverse drill-down windows	The framework and applications can handle different formats display in the drill-down windows
User Interface	Menu additions with components	The framework can handle menu additions to the "Accessories" and "View" menus.
	Time controls (sliders and buttons)	Framework can handle the different user interface devices that change the current scenario time.
	Working window and status display	Framework displays working window and "working" on status bar when doing lengthy background work.

To satisfy the second objective, design and code reuse must be achieved in the component development process. Reuse is calculated by determining how much code from the first application is reused in the second application. Design reuse is analyzed by comparison of the methods used in the various components of each type. Complete classes are not used, but if a majority of the methods are reused, then the design reuse is high.

To fulfill the visualization portion of the objectives, this AFM visualization research uses techniques that show data context and provide micro and macro views of the data. The use of these techniques is compared to previous analysis efforts that use desktop spreadsheet routines. The amounts of data processed and level of detail available to the analyst are the key comparison factors for the two techniques.

3.1 System Architecture

The overall system architecture falls into the heterogeneous architecture category as described by Garlan and Shaw [20]. The architecture is a combination of object-based, batch-sequence, and layered architectures. This combination of architectural styles enables the system to take advantage of object-orient benefits, maintain execution control, and optimize the use of memory, while keeping the user from being overwhelmed with details. By using an object-based architecture the design supports abstraction, polymorphism, and inheritance, raising the potential for reuse. Objects make it easier to isolate different functions and package them into self-contained units supporting component-based design for the first objective. An execute method associated with each visualization component and framework module provides for common communication between the modules in the framework. This "execute" method ensures the framework can control the execution of the components, optimizing memory usage. This approach is similar to that used in the pipe and filter architecture of *vtk* [19]. The architecture is layered, because each module in the framework represents a type of client-server activity. The

AFM visualization system's layers follow the pattern of the ISO OSI network layer stack [2]. For the system, data management is the lowest layer, the graphics rendering is the middle layer, and the user interface is the top layer. Figure 7 is a depiction of this layered concept.

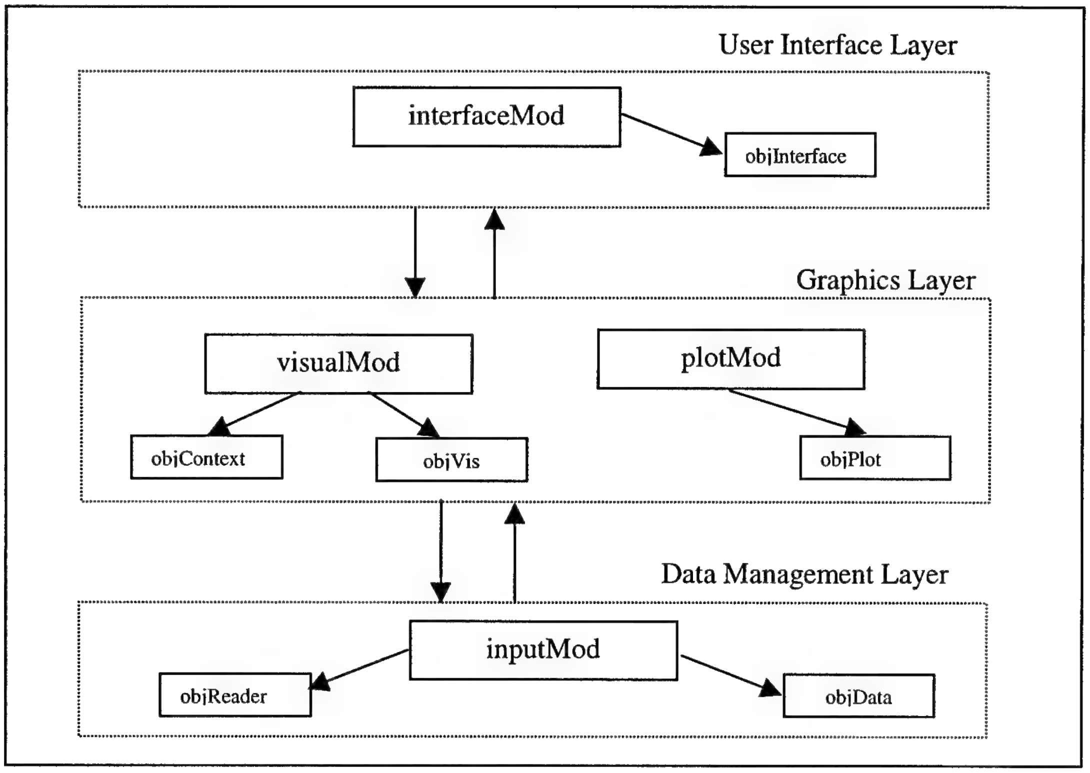


Figure 7: Framework Object Diagram

3.2 Component/Framework Interface Issues

The first objective requires component-based functions to work with this stable-underlying framework through defined interfaces. There were three designs for the component to framework interface reviewed by this research: dynamic plug-ins with user inclusion and execution, programmer inclusion with user directed execution, and programmer inclusion with system controlled execution. In the dynamic plug-in approach the system knows nothing about the components at startup. The user adds the components dynamically at runtime, just by starting them external to the system and pointing the system to them. In the second approach the programmer lists all the components in a container class that the programmer links into the system's executable. The user picks and chooses which components to execute. In the third approach, the programmer includes the components in `main()` and links them into the executable. The difference from the second choice is what and how the user selects functionality. In this third approach the user selects functions not components. A function in this context represents a complete data representation from data input to display.

The disadvantage of the first approach is the overhead the system requires to get the framework access to the components and vice versa. Communication requires sockets or some other external protocol, which would limit the amount of parameter and state information that would be exchanged. For this reason, this approach was not considered to be appropriate for the research.

The second approach minimizes execution size and memory, but greatly increases frustration for the user. If the container class lists all possible reader, data, interface, plot, and visual components, the developer has to convey the relationship between them to the user, or the user would have to know that information *a priori*. For example, if the user adds a new visual component, the user has to also select the reader and data objects supporting that component. Requiring the users to know component dependencies distracts from their analyzing the data.

The third approach manages components for the user, so it was selected for this research. The difference between this approach and the second choice is the absence of the user selecting individual components. The user now selects from the list of data representations presented as menu items that the interface components provide. When the user selects an item the interface component sets the state so the management modules activates or updates the appropriate data, reader, visual, and plot components.

To implement this third approach the application developer selects the components for inclusion in a given executable by including them in the `starter` file's `main()` method. The `starter` file is the reused "main()" for the framework, discussed by Fayad [7], earlier in the methodology discussion in section 3. Inclusion in the `starter` file links the components into the executable, so the system knows about them. The framework controls when the system calls the component's `execute()` or its equivalent method.

This approach satisfies the framework's objectives of making the data representations component-based, simple, and controlled. The application developer has ultimate control over which components he or she includes and excludes from an instance of an application and the framework controls execution in its “inversion of control” [7] discussed earlier in section 3. The memory is optimized because it only contains those processes and data objects the current user selections require. This approach also allows the component-to-framework interface to be defined by base object classes, instead of a socket or other interface. These interface objects are depicted in Figure 7 with their respective layer managers.

3.3 Design Overview

The entire system is divided into two parts, which both use object-oriented class implementations. The first part is the underlying modules and interfaces of the framework. The second part is the component portion, which implements specific applications. Because the system has two parts, the interfaces and interactions between them are critical to the component-based success of the system. This interfacing of the framework and component objects forces the overall system design to transcend both parts of the system.

The system framework's high level design is derived from the *Swift-3D* prototype developed by AT&T [12]. The *Swift-3D* design uses three modules--data collector (data manager), aggregator (visualization manager), and visualization interface (user interface manager). Initial design work on the AFM visualization research validated the need for each of these managers, but also revealed the need for a fourth. Because the AFM visualization research has a goal of letting the user select an entity and get amplifying data on that object, a fourth module is needed to manage the pop-up windows containing drill-down information. Management of these windows is significantly different from the main window, so the visual manager could not manage both.

The four main objects for the AFM visualization framework are an input module (inputMod), visual module (visualMod), interface module (interfaceMod), and plot module (plotMod). Figure 8 shows a UML-based object model of the framework modules as well as the abstract base classes and application components.

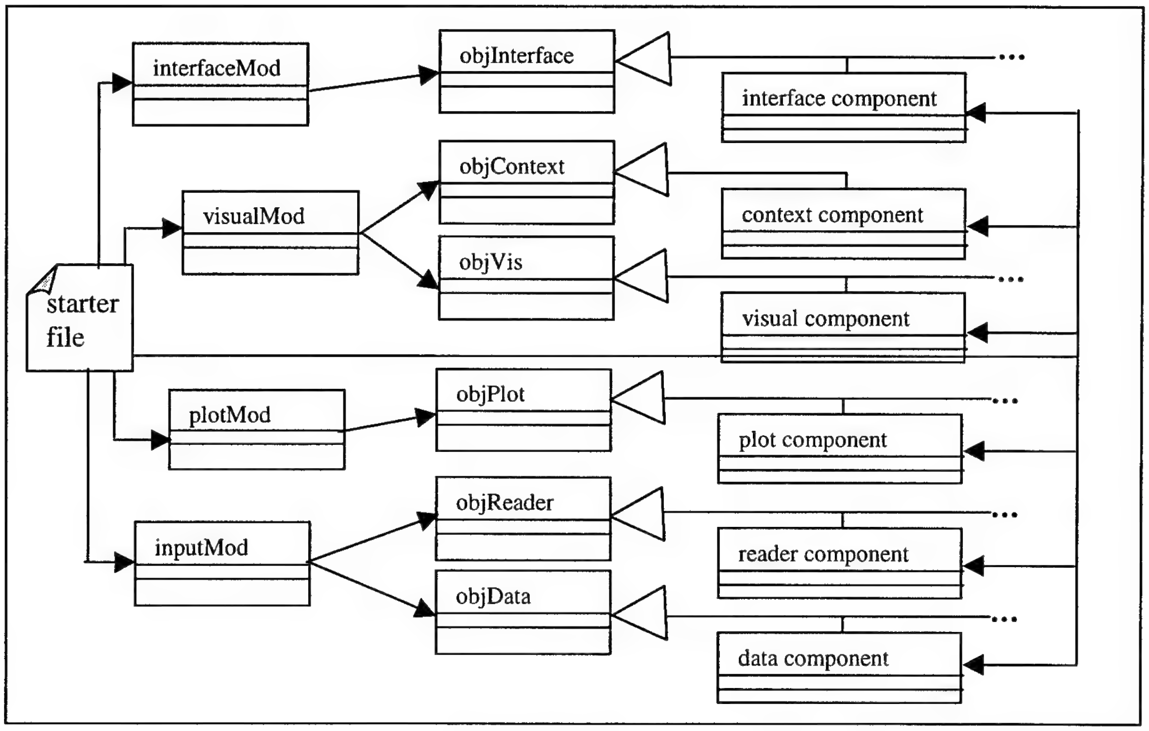


Figure 8: System Object Model

There are six abstract base classes utilized for the framework to component interface. All of the classes in Figure 8 that start with "obj" are abstract base classes. The base classes maintain lists of the components derived from them and provide virtual functions for the components to implement. The base classes are unique to each type of application component and framework module they interface with, but there are still common methods and attributes in these classes. Table 2 provides a description of these common attributes and methods. The virtual methods provided by each base class are discussed in the next sections.

Table 2: Base Class Common Attribute and Method Descriptions

Attribute Name	**Description**
char * objName	Character string containing components name identifier
<object> * <objects>	List of pointers of the object's type. This is the list of child components that each interface object maintains.
Method Name	**Description**
getObjName()	Returns the components name identifier objName
setObjName()	Sets the components name identifier and saves it in objName
addObject()	Adds the calling object to the base class's list of child components.
Find()	Takes in the name of a child component and returns a pointer to that child component if it is found in the base objects list.

There are six categories of components, one for each abstract base class in the framework. The component categories are readers, data, visuals, context, plots, and interfaces. These components implement the virtual functions of their respective abstract base class. For this AFM visualization research the components' names replace the "obj" portion of the abstract base class's name. For example, the military "base" visual component would be `baseVis`. With this naming scheme components can be easily matched to a base class and management module. For example, `baseVis` is easily associated with `objVis` and in turn the visual module.

3.4 System Design

The next sections explain each layer in more detail. The management module or modules for each layer are discussed as well as the abstract base classes unique functionality used by the module. The layer's application components used by a developer to implement an application are also discussed with each layer.

3.4.1 Data Management Layer

The Data Management Layer manages the data retrieval and storage for the rest of the system. The management module for this layer is the input module implemented as `inputMod`. Data input for the system is done by reader components that read the appropriate input file and then create and populate the data objects. Data components contain the data structures that store the data. The input module manages the reader and data components with two abstract base classes. Figure 9 shows the relationships between these objects. There is one abstract base class for readers and one for data objects.

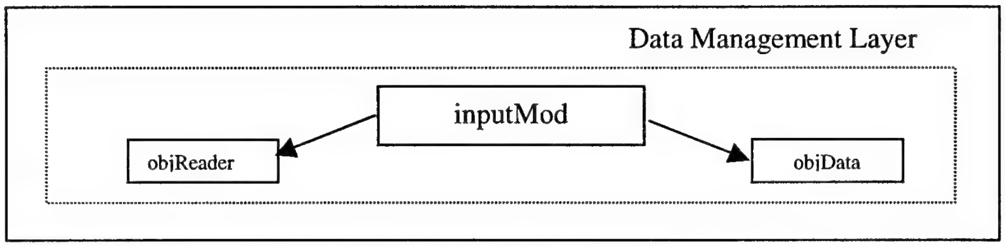


Figure 9: High-Level Data Management Layer Object Diagram

Having a central data manager provides a single point for all other objects to request and get data. This single manager design helps to ensure that multiple copies of a data set are not read into memory and only those levels needed are in memory. The design minimizes memory usage by controlling the presence of populated data objects. Having a data manager also hides the other components from the details of reading the files and populating the data.

3.4.1.1 Input Module

The input module is implemented as `inputMod` and uses two abstract base classes to interface with the application components. Figure 10 shows the detailed object diagrams for

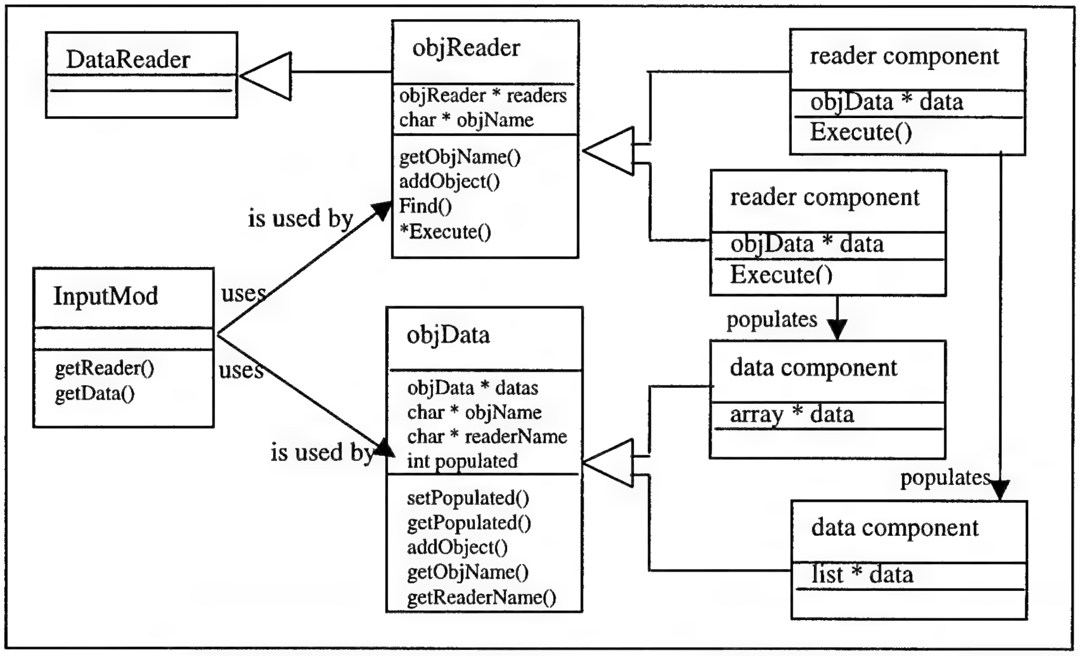


Figure 10: Input Module Object Diagram

these objects. The input module provides two interfaces for the other layers: a `getData()` method and a `getReader()` method. These two methods provide the means for an object to request data from the input module by passing a name and data level to the `getData()` method. For the `getData()` method, if the requester asks for a data object populated at a particular percentage, the input module returns a pointer to the data object. If the data is not sufficiently populated, the input module retrieves the required reader and calls its `execute()` method. If a component in the visualization layer calls `getData()`, it only needs to pass the data object name and required population level. The input module interfaces with the other objects in the layer and returns a pointer to the appropriate data object if it is available. By returning pointers to the objects the system maintains only one instance of the object.

3.4.1.2 objReader Base Class

The `objReader` base class provides the common base class methods and the `DataReader` methods as well. `DataReader` is a *vtk* class that has methods for reading everything from floats and integers to strings and characters. The `objReader` class inherits from `DataReader`, to provide the reader components with common local methods for reading input data. This is another way to reduce duplication and increase reuse in the framework.

3.4.1.3 Reader Components

The reader components read the data files and populate the data objects. Each reader component is a child class inheriting from the `objReader` class. The input module executes the reader by calling the virtual `execute()` method from the `objReader`. Since the `execute()` method is a virtual method each component must implement this function. Each reader is unique, but in general the `execute()` method opens the input file, reads the data and populates the data structures in the data objects. The amount of data reading done by a reader is left up to the implementation of each reader. For some data files a call to the `execute()` method causes the reader to read the entire file. For other readers the integer value passed to the `execute()` method represents a limit to the amount of data read during an execution. The input module passes on the requester's populate level; what that represents to the reader, the data, and the requester of the data is independent of the input module. The framework will only call the reader if someone requests the data and the data object's population is less than what was requested. This approach gives the reader and data component developer control over implementation, while enabling the framework to maintain its generic, but reliable behavior.

3.4.1.4 objData Base Class

The base class for the child data objects (data components) is `objData`. The `getObjName()` and `getReaderName()` methods return the child's name and it's corresponding reader's name. Child objects also maintain a `populated` attribute that reflects how much data has been read. The populated attribute is an integer, so it can be used in one of several ways for a particular data object. If the data is populated in an all or nothing manner, a zero or one is used for a yes or no population status. If however, the reader can populate the data incrementally it can be used as a percentage value like 90 to represent 90% populated. The actual representation is left to the component designer.

3.4.1.5 Data Components

The data components are objects containing data needed by other parts of the system. The designer of the data object determines the data structure most appropriate for the data component. The data objects inherit from the `objData` abstract base class. The data component does not have an `execute()` method, so the input module interface with these components is the `objName` and `populated` attributes. The input module uses these attributes to find the data object and compare its population levels with the needs of the requester.

3.4.2 Graphics Layer

The graphics layer manages the OpenGL rendering routines in the main window and drill-down windows. As mentioned in section 3.1 this layer has two managers to handle the displays. The first manager is the visual module implemented as `visualMod` in Figure 11. The visual module manages the context and data representations in the main window. This requires knowing the availability and status information of each visual component. Like the input module, the visual module interfaces with two abstract base classes `objContext` and `objVis` in

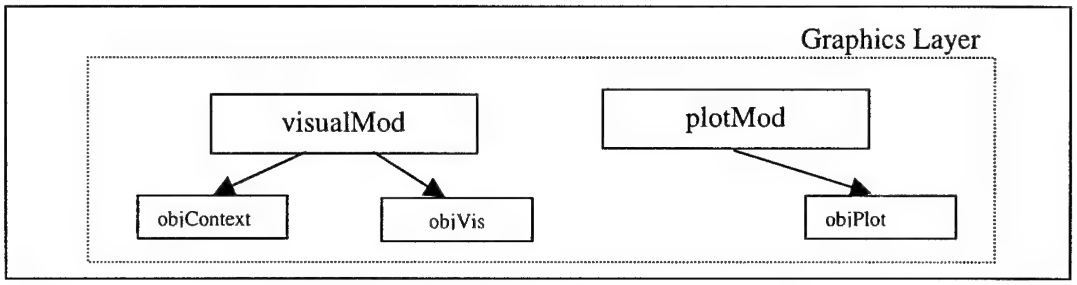


Figure 11: High-Level Graphics Layer Object Diagram

Figure 11. The second manager for the Graphics Layer is the plot Module implemented as plotMod in Figure 11. The plot module manages the drill-down windows for the system. The plot module only interfaces with one base class, `objPlot`.

By utilizing a central visual module, the interface module only has one callback object for the main window. The central visual module enables a single object to coordinate the interaction of the user interface zooming and rotating with the context component and picking objects with the other visual components. A central plot module provides the interface module with a single object to handle the entire collection of drill-down requests made by the user.

3.4.2.1 Visual Module

The visual module is implemented as visualMod and uses two base classes to manage its components. Figure 12 is an object diagram for this portion of the graphics layer. The visual module has a large number of methods, but a majority of these methods are graphical user

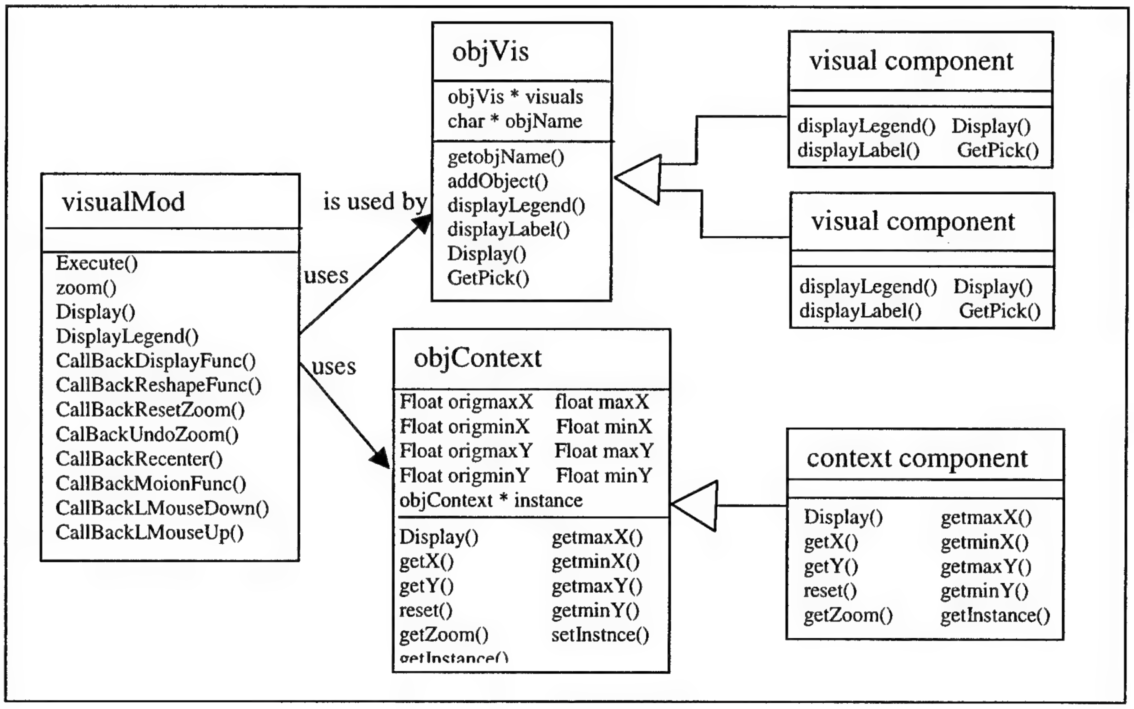


Figure 12: Visual Module Object Diagram

interface event callback functions. Callbacks are methods that register for specific events with the framework's event loop. When that event occurs, the callbacks registered for that event are called. Table 3 gives a brief functional description of the callbacks in the visual module.

Table 3: Visual Module's Callback Methods Descriptions

Method Name	Description
`CallBackDisplayFunc()`	Manages the calls to the three display methods of each of the visual components and draws the zoom box
`CallBackReshapeFunc()`	Updates the main window renderings and adjusts the size and scaling parameters when the window size or shape changes
`CallBackResetZoom()`	Resets the context and data display scales to their original settings
`CallBackUndoZoom()`	Undoes the last zoom operation input by the user
`CallBackZoom()`	Zooms in or out the context and data displays based on user input
`CallBackRecenter()`	Shifts the context and data display to center on the user input point
`CallBackLMouseDown()`	Starts drawing the zoom box when the user pushes left mouse button
`CallBackLMouseUp()`	Stops drawing the zoom box when the user releases the mouse button
`CallBackMotionFunc()`	Updates current location of moving mouse while button is down

The visual module's `execute()` method acts like an initialization by starting the rendering, establishing the context drawing, and defining the window sizes. The `pickObjects()` method is called by the interface module when the user selects an object from the main window display. This `pickObjects()` method calls each visual component's `Display()` method in SELECT mode to see if the mouse coordinates hit anything the component is displaying. OpenGL has two modes for rendering graphics primitives RENDER and SELECT. The RENDER mode sends the images to the screen. The SELECT mode sends the images to a buffer for processing. The processing in this case is checking the mouse click point with the existence of a graphics item. If an item is selected, the `pickObjects()` method returns the information the plot module needs to process the selection.

Another task accomplished by the visual module is rendering the legend. The legend, Figure 13, is a gray box in the lower left corner of the display. To render the legend the visual

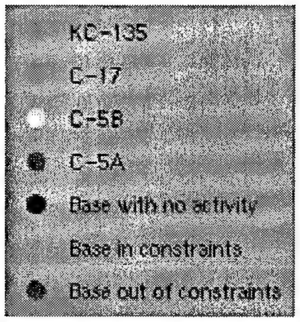

KC-135
C-17
○ C-5B
● C-5A
● Base with no activity
Base in constraints
● Base out of constraints

Figure 13: Picture of Legend (Plate 2)

module calls each visual component's `DisplayLegend()` method. The call passes the location of the lower left corner of the current line and the height of a line in the legend. The component's method returns the number of lines used, so the visual module can adjust the lower left corner for the next component. This process enables the visual module to continuously add other component's items to the legend with only minimal knowledge of what is being displayed.

3.4.2.2 objVis Base Class

The `objVis` class is the abstract base class for all main window visual components. The main window canvas is the map and ocean area of Figure 14. Unlike the other base classes, the `objVis` class lacks a single `execute()` method. Instead, there are three such methods

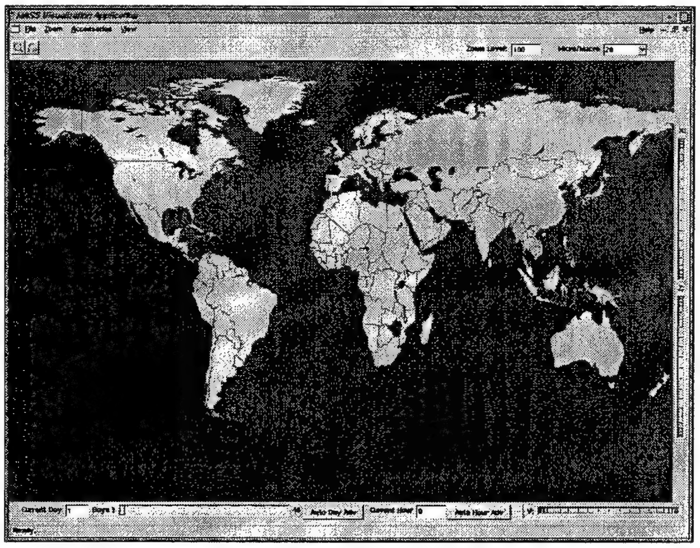


*Figure 14: Screen Capture of Main Window with Map Context (*Plate 1*)*

`Display()`, `DisplayLabels()`, and `DisplayLegend()`. The visual module calls these three virtual methods in the child objects based on the user's current selections. It is the responsibility of the component's implementation to execute the specific call.

3.4.2.3 Visual Components

The visual components convert the input data to computer graphics primitives for display on the main window. The visual components provide implementations for three virtual functions in the `objVis` base class: `Display()`, `DisplayLegend()`, and `DisplayLabel()`. The `Display()` method executes the OpenGL commands to render the objects shape and color for the display of the data. The "Legend" menu item from the "Accessories" menu activates the visual module call to `DisplayLegend()`. This method renders a representation of the visualization with a description in the legend area of the display (Figure 13). The "Labels" command from the "Accessories" menu activates the call to `DisplayLabels()`. This method displays labels for the data. The label itself is left up to the designer of the component. For the implementations in this AFM visualization research, the labels are text of the ICAO abbreviation.

The visual components query the interface module to see if the menu item controlling their display is selected or not. To accomplish this visual components call the interface module's `getStatus()` method. These method returns true or false based on the current status. By calling the interface module for the status of the interface object, there is no component to component communication, keeping with the idea of using the framework interface exclusively.

3.4.2.4 objContext Base Class

The `objContext` class is the abstract base class for the context components. Figure 14 is an example of a global map context. The `objContext` class is different from the other base classes, because it does not maintain a list of context objects and it has many more virtual methods. No list is maintained, because there is only one context component for an application. The context component object uses the `setInstance()` method to register as the single context `instance`. The other methods like `getX()`, `getY()`, and `reset()` are all virtual methods used by the visual module to convert from screen coordinates to the context-based coordinates. These conversions are necessary for zooming, scaling, re-centering, and picking.

3.4.2.5 The Context Component

The context component provides the system with the contextual background display for the visualizations. The designer of the context object determines the context for the visualization. The context component inherits from the abstract base class `objContext`. The component's implementation of the virtual `Display()` method is similar to the `Display()` method of the visual components. The `Display()` method gets its data from the input module and then renders the graphics primitives for the context object. The visual module uses a graphical display list to render the context. The display list maintains the graphical commands in memory, thereby supporting rapid redraws, since the context object's graphics primitives do not change.

Beyond the `Display()` method, the context component has little similarity with other visual components. The methods `getX()` and `getY()` both take in a screen coordinate and return a context-based coordinate. For the map context in this AFM visualization research they return a latitude and longitude value. The `reset()` and `getZoom()` methods adjust the current corners of the display when the user resets to the original or zooms a certain percentage. The visual module calls these methods when the user draws a zoom box or selects an item from the "Zoom" menu. The visual module calls the context component's "get<Attribute>" methods to access the coordinate values after the previous `reset()` and `zoom()` methods finish executing.

The context component inherits two groups of attributes (current value and original value holders) that represent the coordinates of the four sides of the display. The four starting with "orig" in the name, store the starting values. The other four attributes are used by the methods discussed in the previous paragraph to change as the user changes the view of the scene. For example, the `zoom()` method changes the second group by the percentage of zoom selected by the user.

3.4.2.6 Plot Module

The Fox graphical user interface libraries [8] provide the windows and interface devices in this AFM visualization research. In the Fox hierarchy of window management, managing the main window is separate from managing the pop-up windows. This separation forces the introduction of another visualization layer module in the framework: the plot module, implemented as the `plotMod`. This plot module manages all the component objects providing drill-down information on a picked object. A common thread between all of the objects in this AFM research is they are all "plots" of data; hence the name "plot module".

The plot module implements a tab book (Figure 15) for the drill-down window when a user selects an object for amplification. It is called a tab book, because it is like the pages of a

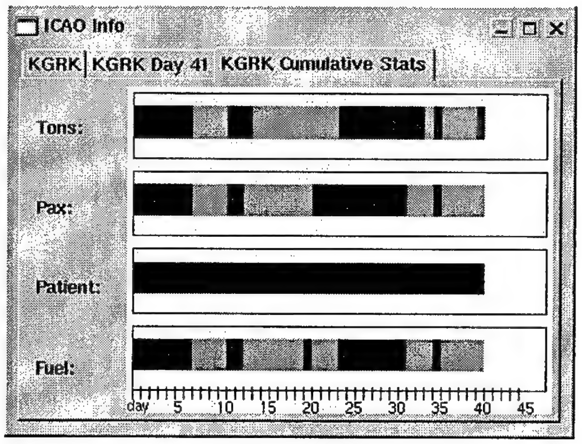


Figure 15: Picture of Plot Module Tab Book (Plate 3)

book that the user looks through by selecting the tab of interest. When a tab is selected, that page's information is brought to the front of the display. This design optimizes screen real estate and provides the user a means to control what information is shown. The plot module uses a single abstract base class called `objPlot` to manage the plot components. Figure 16 shows the relationship between `plotMod` and `objPlot`.

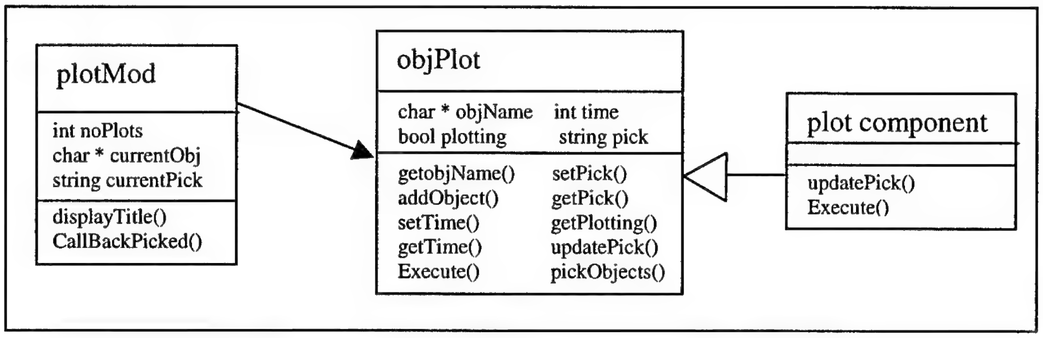


Figure 16: Plot Module Object Diagram

The plot module provides the interface module a single callback for all drill-down data requests. The `CallBackPicked()` method checks to see if a component matching the picked object's handle is currently showing or if a new window needs to be opened. For example, if the user picks an ICAO from the main window, the interface module calls `CallBackPicked()` and passes it the object that was picked (ICAO), the current time, and the picked information string. The plot module loops through the list of components to check if any are registered for the handle "ICAO". If a component is registered, the plot module checks to see if the component is currently plotting. If the component is plotting, the plot module updates the component's current data. If a component is not plotting, the plot module sets up a new tab book for the component

3.4.2.7 objPlot Base Class

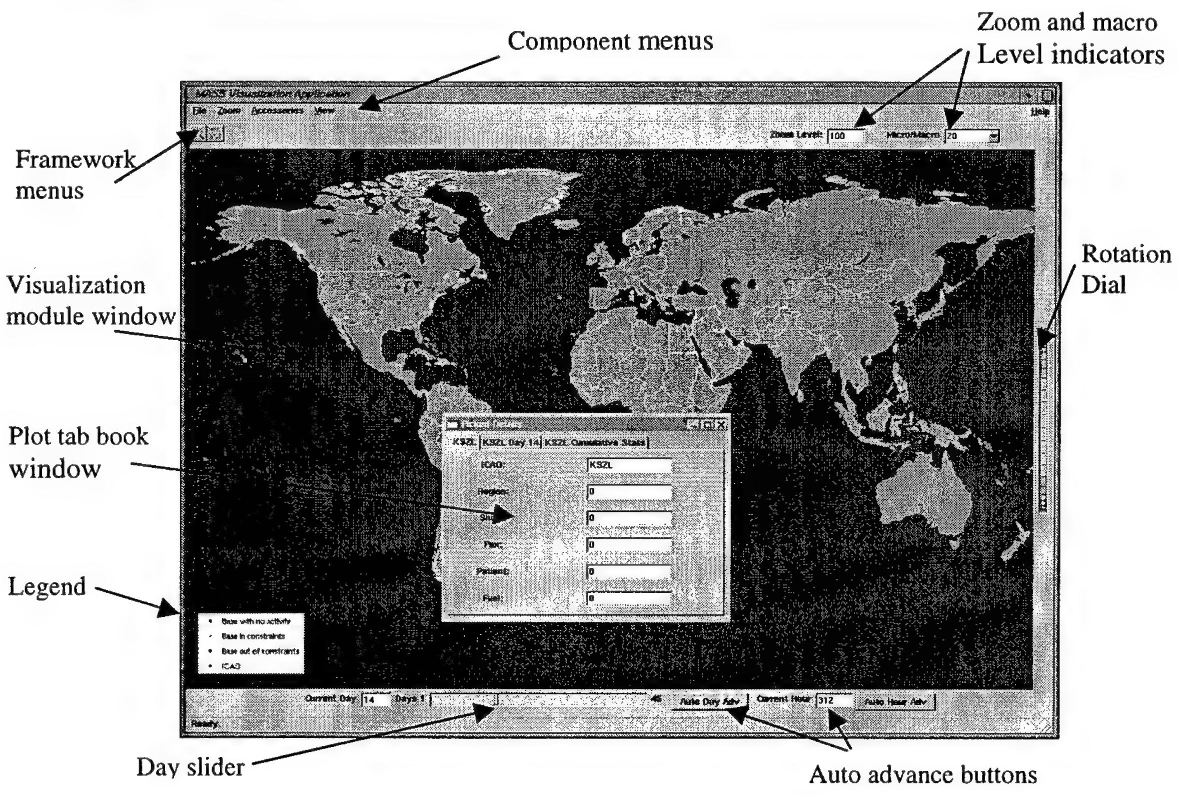
The `objPlot` class is like the other base classes, except the child class names have two uses. This child object name is not only the means for the plot module to identify the component, it is also the message handle passed by a picked object. The `time` attribute contains the scenario hour the object was picked. The `currentPick` attribute contains the information the component uses to get the additional data needed to display the drill-down information. The data type of the `currentPick` attribute is a string, so the plot module can pass anything encoded in a string to the components. The `currentPick` string can contain anything like an aircraft tail number, a base name, or even a cargo description. For example, if the user picks an aircraft from a visual component, the component includes the aircraft's tail number and the current base's ICAO in the string. The values the string contains are the keys the plotting component needs to find the data in the appropriate data structure. The component developer determines the string's contents. Using this string enables the interface to remain common among all plot components, but yet individualized for each data set's keys. Whether the key is a database key, a single character, or even an index number, the plot module's interface remains the same. The Boolean attribute `plotting` conveys the current plotting status of the component. The `plotting` attribute is true if the object is currently displayed and false if it is not being displayed.

3.4.2.8 Plotting Components

The plotting components render drill-down information in the pop-up windows when a user picks an object. Each plotting component is a child class inheriting from the `objPlot` class. The plot component implements the virtual methods `execute()` and `updateTime()`. The plot module sets the `currentPick` attribute and calls the `execute()` method when the plot component is being displayed in a new window. The `updatePick()` method is similar to the `execute()` method, except the display does not have to be created. Since the plot component is already displaying a tab book, the plot module updates the `currentPick` and `time` attributes and the `updatePick()` method queries for the new data and updates the tab books values.

3.4.3 User Interface Layer

The User Interface Layer creates the windows, manages the user interface devices, initiates the plot windows, and coordinates all of the input from the user. The interface module uses the Fox interface libraries to provide a Fox main window, sub-windows for plotting and most importantly, an event loop for user interaction. Figure 17 shows a screen capture of the display and points out the main interface items. The main interface window contains the majority of the user interface items. Default framework menus, rotation dials for tilting the display for 3-D viewing, day and hour controls, and text fields of the zoom level and macro cutoff levels are all part of the main window interface. The time controls for the user include a day slider and auto-advance buttons for automatically advancing the day and hour to animate the visualization through the scenario.



Zoom Level: 100 Micro/Macro: 20

ICAO:	KSZL
Region:	0
Sh	0
Pax:	0
Patient:	0
Fuel:	0

Current Day 14 Current Hour 312

*Figure 17: Picture of User Interface (*Plate 4*)*

Interface components add menu items to the "Accessories" and "View" menus that are part of the framework. The components add to the interface by inheriting from the abstract base class `objInterface`. Figure 18 shows the high-level object model for the interface layer.

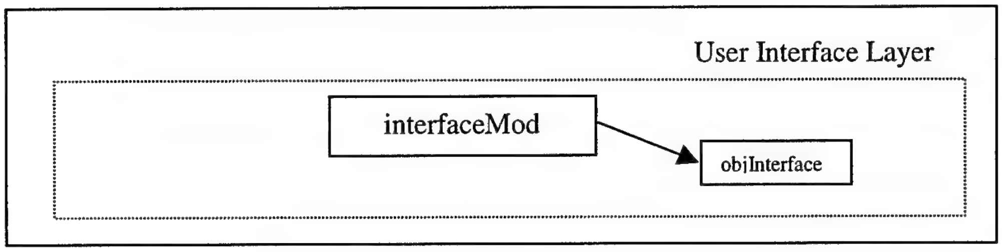


Figure 18: High-Level User Interface Layer Object Diagram

3.4.3.1 Interface Module

The interface module is implemented as `interfaceMod` and uses a single abstract base class to interface with the application components. Figure 19 shows a detailed object module of

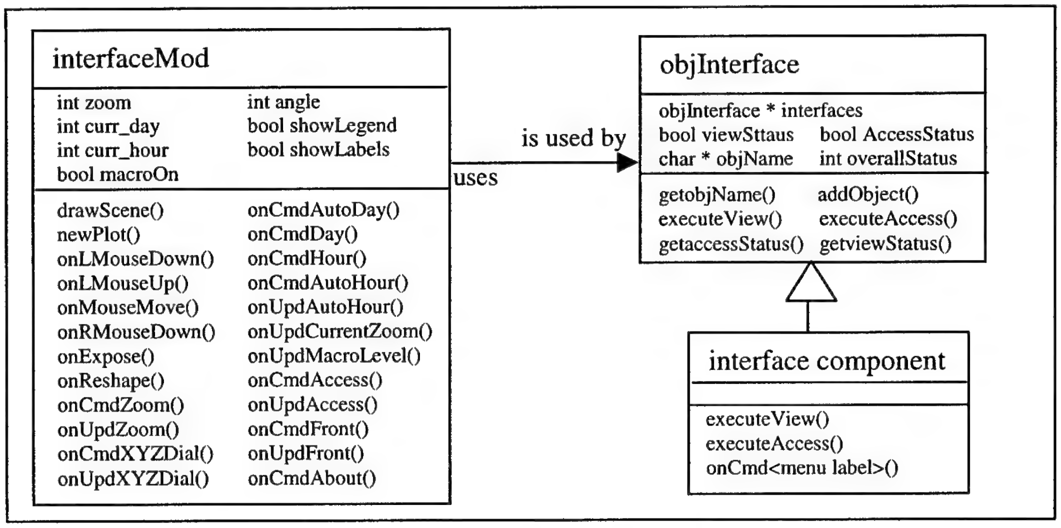


Figure 19: User Interface Framework Object Diagram

this relationship. As is typical in any user interface class, a majority of the methods in the interface module are user interface callbacks. These callbacks implement the functions registered with the Fox event loop to handle specific user interface events. Table 4 lists and describes each of these callback functions found in the interface module.

Table 4: Interface Module's Callback Methods Descriptions

Method Name	**Description**
`onExpose()`	Calls DrawScene() any time the main window is being uncovered, at start up and when another windows is move from on top of the Fox window.
`onReshape()`	Updates the window attributes when the window size or shape is changed
`onLMouseDown()`	Handles left button events calling the visualMod's CallBackLMouseDown
`onLMouseUp()`	Handles left button events calling the visualMod's CallBackLMouseUp
`onMouseMove()`	Handles left button events calling the visualMod's CallBackMouseMotion
`onRMouseDown()`	Processes the picking of objects by the user calling the visualMod's PickObject() method and then the NewPlot() method if one is picked.
`onCmdZoom()`	Handles selection of "Reset Zoom" button, sets the zoom level back to 100% and then calls visualMod's CallBackResetZoom()
`onUpdZoom()`	Monitors zoom levels to gray out "Reset Zoom" button when not zoomed
`onCmdFront()`	Handles user selection of the "Front" button, which returns all of the x,y, and z rotation angles to zero, sot he display is straight up and down again.
`onUpdFront()`	Monitors angles to gray out "Front" button when display not rotated
`onCmdAbout()`	Displays the About pop-up window when the user selects the "Help|About" menu.
`onCmdAccess()`	Handles selection of the "Accessories" menu items and sets `showLegend()` and `showLabels()` attributes.
`onUpdAccess()`	Monitors "View" menu overallStatus, because "Accessories" menu is grayed out if no "View" items are currently selected.
`onUpdCurrentZoom()`	Updates the "Zoom Level" text field when the level changes
`onUpdMacroLevel()`	Handles the user selection zoom level to switch from macro to micro view
`onCmdXYZDial()`	Updates the angle values when the user rotates the dials to rotate the scene
`onUpdZYXDial()`	Resets the dial settings after the user resets the rotation of the scene
`onCmdAutoDay()`	Steps through the days one at a time until the end of scenario or user input
`onCmdDay()`	Handles user inputs from the day slider and updates the curr_day attribute
`onCmdHour()`	Handles inputs from the day slider and updates the curr_hour attribute
`onCmdAutoHour()`	Steps through the hours by ones until the end of scenario or user input

The `newPlot()` method in the interface module communicates with the plot module's `CallBackPicked()` method. The interface module calls the `CallBackPicked()` to determine if a new pop-up window needs to be created or if the current plot windows are sufficient. As discussed in Section 3.4.2.6, if the user selects an object that is not currently plotting, the plot module creates a new tab book for the component. If the plot module needs a new window the interface module creates a new sub-window and passes it to the plot module for the new tab book for the plot component. Once the interface module creates the window or the plot module returns with existing window for the object, `newPlot()` is complete.

The onUpdScene() method works with the drawScene() method to communicate with the visual module. The drawScene() method calls the visual module's CallBackDisplayFunc() method to force a re-rendering of the scene in the main window. The Fox event loop calls this method to update the scene continuously if no other activity is running. This is a problem with complex scenes like those in the AFM visualization research. To reduce this problem, the interface module uses the onUpdScene() method to respond to the Fox event loop's request for an update. The onUpdScene() method scans the interface module's attributes like angle, zoom, and time to see if any have changed. The onUpdScene() method only calls the drawScene() method when one or more of these attributes have changed. This approach greatly increases the systems performance, by freeing up the processor from constantly re-rendering the scene.

3.4.3.2 objInterface Base Class

The executeView() and executeAccess() methods of objInterface are virtual functions the interface component classes must implement. The implementations are unique to each class, but in general these methods add the menu item to the interface module's "View" and "Accessories" menus and register an ID for the menu item with the Fox event loop. The viewStatus and accessStatus attributes contain the current status of the menu items; either selected or unselected. The static class-wide integer attribute overallStatus tracks how many current menu items are selected. When the user selects a component's menu, it increments overallStatus and when the user de-selects the item, it decrements overallStatus. The "Accessories" menu uses the overallStatus attribute to determine if its menu items should be accessible or grayed-out. The "Accessories" menu is only enabled if the user selects one or more of the items on the "View" menu.

3.4.3.3 Interface Components

As a general rule, the interface components are the smallest components for a new application since they only add menu items to the main window interface. The addition of other interface items such as buttons and dials is an area for future research. The additional menu items give the user more control over what the display presents. The interface components are child classes that implement the `objInterface` abstract base class. The interface components represent the top layer of the hierarchy, the point of interaction with the user. They are the starting points for activating the data representation in the system.

The interface components inherit attributes `viewStatus`, `accessStatus`, and `objName`. The Booleans `viewStatus and accessStatus` attributes are set to true if the respective menu item is currently selected or false if it is not currently selected. The interface components also increment the `objInterface's overallStatus` static attribute when the user selects the menu item. It decrements the `objInterface's overallStatus` static attribute when the user de-selects the menu item.

The interface components also implement the `execute()` virtual method of the `objInterface` abstract base class. In general this `execute()` method establishes the name of the menu entry, the help information for the item, and the callback ID for the menu. The interface component's `onCmd<menu label>()` method is the callback method it registers for each menu item. The callback ID and `onCmd<menu label>()` callback method are the interface to the Fox event loop. Selecting the menu item causes the Fox event loop to initiate the call to this method. This method sets the `viewStatus` or `accessStatus` and `overallStatus` attributes as discussed in the object interface section above.

3.5 Methodology and Design Summary

The methodology for this AFM visualization research is to develop a component-based system that runs on an application framework. This AFM visualization application framework is based on a four module layered architecture. The main interfaces to the framework are the six abstract classes for the components. The components implement the virtual methods of the abstract base classes. The programmer, by including or excluding the component in the starter file, controls the mixing and matching of functions in an application. The inversion of control by the framework manages the actual execution from there. This methodology supports the meeting of the goals and objectives of this AFM visualization research.

4 *Implementations*

To test the objectives of this research, two applications were designed and implemented. These applications are the source of the metrics presented in the results section of this document. The applications validate the architecture, the framework's interfaces, and the reuse of source code in component development. Their use of the entire framework with different data structures and visualizations was the main reason for implementing these applications for the AFM analysts. This chapter discusses these two applications in terms of reason for selection, data structures, displays, and drill-down capabilities. The chapter concludes with a mapping of these two applications to the success criteria outlined in Chapter 3 of this AFM research.

4.1 Daily Airfield Statistics

The first application tracks daily activity and constraints of an airfield in the area of short tons of cargo, patients, pax (civilian passengers), and fuel. The daily data for each category is found in the summary file produced by AFM for each airfield. The location information for the airfields is found in the f25_1 AFM input file. This initial effort only required development of two readers, one data object, three plot components, and two visual components. This simple data requirement was the reason for selecting this information to visualize in the first application.

Table 5 shows the components comprising this application and the module they support, interfaces they exercise, and other items exercised. For example the `loc_listReader` component is a reader component that implements the `objReader` base class. The `loc_listReader` class uses the `objReader's addObject()` method and implements the `execute()` method called by the `inputMod`. Some of the additional items this class exercises are the `DataReader` object's methods to read floats, integers, and char strings. Another example from the table is the `airfieldcumVis` class, which implements the `objVis`

Table 5: Component Descriptions for Application #1

Name	Type	Framework interfaces	Other items
loc_listReader	objReader	objReader - addObject(), execute()	floats, integers, char
loc_listDataMap	objData	objData - addobject()	"map" data structure
summaryReader	objReader	objReader - addObject(), execute()	large file read
summaryobjData	objData	objData - addObject()	array data structure
mapReader	objReader	objReader - addObject(), execute()	builds multiple data objs
landData	objData	objData - addObject()	large array data structure
waterData	objData	objData - addObject()	large array data structure
mapContext	objContext	objContext - all methods, inputMod - getData()	Display lists and large data sets for graphics engine
baseVis	objVis	objVis - Display(), DisplayLabel(), DisplayLegend() inputMod-getData(), interfaceMod - getStatus()	single point display, view menu, and object picking
airfieldstatVis	objVis	objVis - Display(), DisplayLabel(), DisplayLegend() inputMod-getData(), interfaceMod - getStatus()	micro/macro level display, view menu, object picking, colors, and day-based time
airfieldcumVis	objvis	objVis - Display(), DisplayLabel(), DisplayLegend() inputMod-getData(), interfaceMod - getStatus()	statsVis items and does data processing by correlating data.
basePlot	objPlot	objPlot - execute(), updatePick(), and inputMod - getData()	tabbook, Fox textfields
statsPlot	objPlot	objPlot - execute(), updatePick(), and inputMod - getData()	tabbook, Fox textfields, and OpenGL based bar charts.
cumPlot	objPlot	objPlot - execute(), updatePick(), and inputMod - getData()	statsplot items and plot window-based obj picking
baseInterface	objInterface	objInterface - executeView()	Fox event loop, view menu
statsInterface	objInterface	objInterface - executeView()	Fox event loop, view menu
cumInterface	objInterface	objInterface - executeView()	Fox event loop, view menu

abstract base class. This class exercises the `Display()`, `DisplayLabel()`, and `DisplayLegend()` methods called by the `visualMod`. It also exercises the `inputMod's` `getData()` method and the `interfaceMod's` `getStatus()` method. Additionally, `AirfieldcumVis` populates the view menu, provides different micro and macro level views, uses colors, and is also the only visual component to do data processing by correlating the data received from the `summaryobjData` data object.

The application is based on two AFM files, the f25_1 file and the summary file. The f25_1 input file is read by the `loc_listReader` and contains a list of the International Civilian Air Organizations (ICAOs), their latitude, longitude, and initial constraints in cargo, patients, pax, and fuel. The summary output file is read by the `summaryReader` and lists several attributes of aircraft and airfields on a daily basis. The summary file lists the ICAO designator and the daily activities and constraints in the four categories of cargo, patients, pax, and fuel, for each ICAO.

4.1.1 Data Structures

For the daily airfield statistics application, the queries into the data was based on an ICAO designator and a particular scenario day. This need for a two-key index for the data meant there needed to be a multilevel data structure for the ICAO location data. Figure 20 gives a brief depiction of the multilevel data structure. The day-based array was chosen, as the top level of the

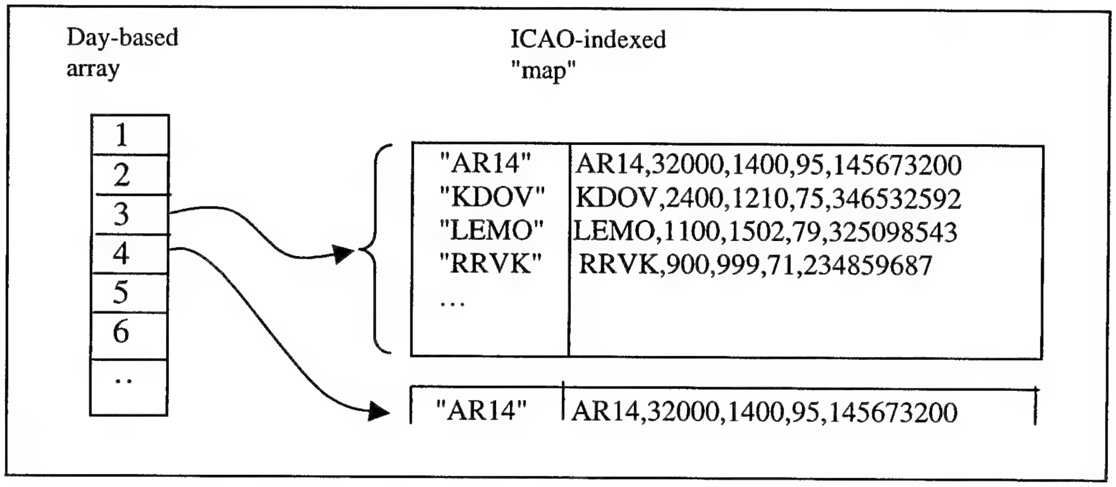

Day-based array
ICAO-indexed "map"
1
2
3
4
5
6
..
"AR14" AR14,32000,1400,95,145673200
"KDOV" KDOV,2400,1210,75,346532592
"LEMO" LEMO,1100,1502,79,325098543
"RRVK" RRVK,900,999,71,234859687
...
"AR14" AR14,32000,1400,95,145673200

Figure 20: Application #1 Data Structure Diagram

data structure, because the queries in this application are day-to-day based not ICAO-to-ICAO based. If they were ICAO-to-ICAO based, the data structure would have had the ICAOs at the top and an array under each containing the different days' data. For the second level of the data structure, the C++ Standard Template Libraries (STL)[13] contain a "map" data structure whose content type and index key are programmer defined. For this application the index is defined as the ICAO four-character designator and the data structure holds the ICAOs' locations and constraint information.

As the user steps through the days in the application the visual components increment through the top-level array. Then to iterate through the ICAOs with status for that day, the map's `iterate()` method is used. When drill-down information is needed for a particular airfield, the day is used to index the array and the airfield's ICAO designator is used to access the data in the second level "map" directly.

4.1.2 Visual Display

The display of the daily statistics was implemented in two forms with a micro and macro view for each. The first form (Figure 21) shows the airfield's statistics for the current day. The second form (Figure 22) shows an accumulation of the statistics from the first day. Both forms use the same color-coded approach of blue symbols for no activity for the day, green for activity within constraints, and red for activity that exceeded constraints.

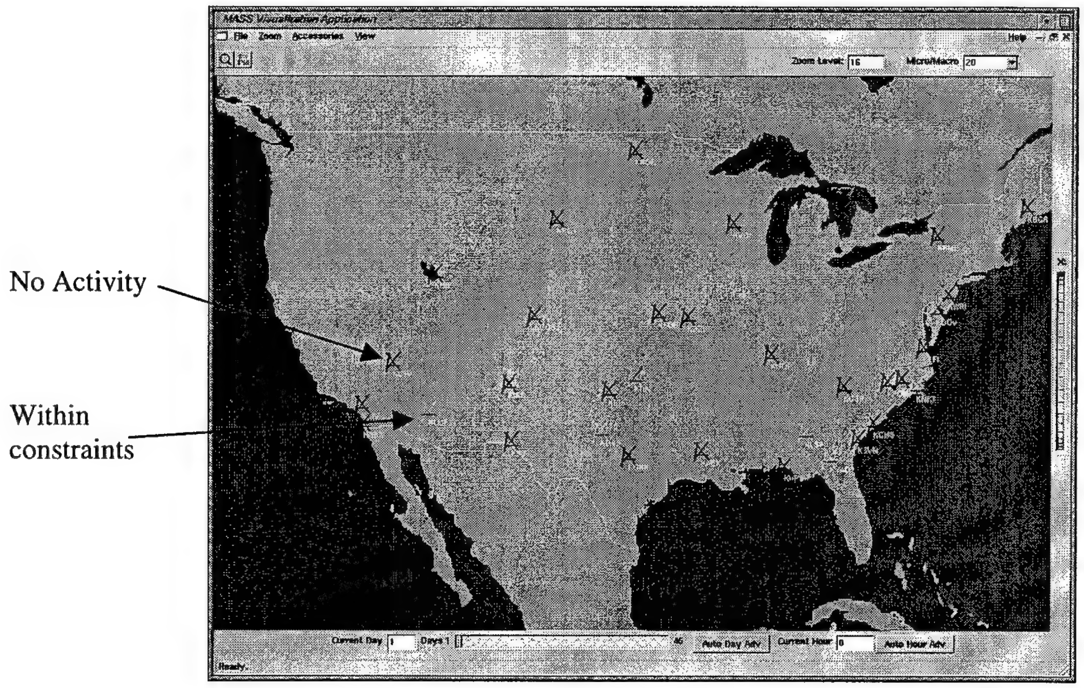


Figure 21: Application #1 Micro View (Plate 5)

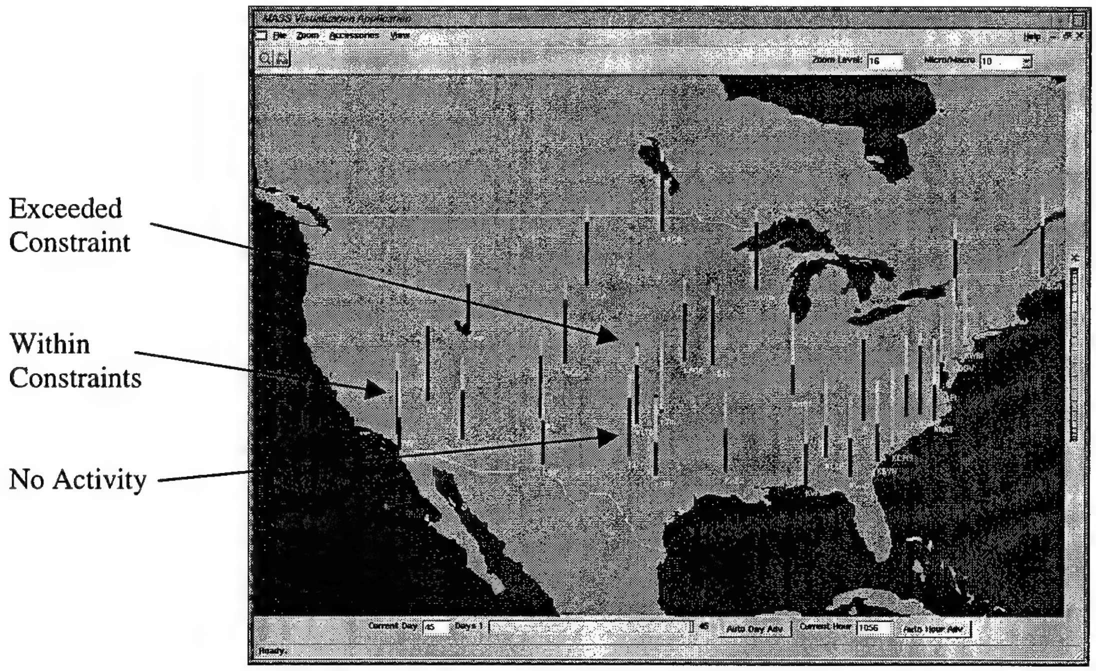


Figure 22: Visualization #1 Cumulative View (Plate 6)

The daily form uses a color-coded box representing the airfield on the map (top of Figure 23). At the macro level this box represents a summary of the activity in the four categories. For

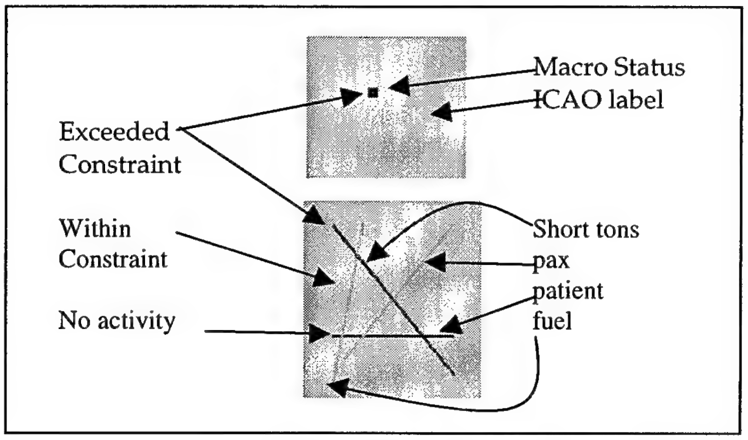


Figure 23: Statistic Symbols (Plate 7)

example, in Figure 23, the box is red meaning at least one category exceeded constraints for the day. For the micro view a four-runway airfield-like glyph (bottom of Figure 23) is used to represent the four individual categories of activity. For each day the respective runways are color-coded to show the status. In Figure 23, the micro view shows cargo exceeded constraints, pax and fuel were within constraints, and patient was inactive. For the status of a single data category in the scenario this provides the analyst a great deal of information in one picture. However, as the analyst changes the day for simulation playback, the symbols appear to merely flicker, preventing any trend analysis or comparison from day to day. Therefore, an additional display of the data was needed, so the cumulative visual component was developed.

The cumulative display is a total of the daily status, providing the trend analysis and comparison the analyst needs. This cumulative form uses the same symbols as the first, except the current days statistics are added to the previous day's statistics. The analyst can now get a 3-dimensional bar-type glyph growing out of the airfield on the map representing the statistics. Figure 24 shows the micro and macro view of the cumulative status. In Figure 24, the ICAO

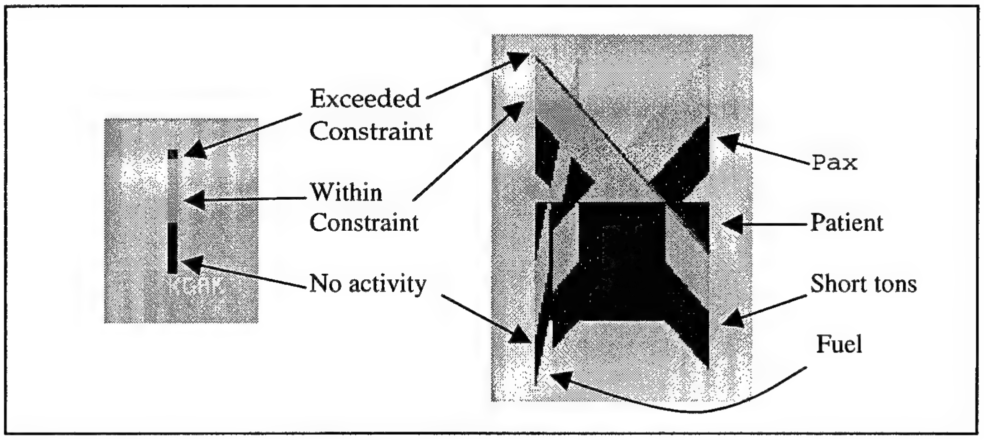


Figure 24: Cumulative Symbols (Plate 10)

exceeded constraints one day for short tons of cargo. When a day's status is added to the cumulative symbol, like status is grouped together for easier comparison between ICAOs. This allows the analyst to compare airfields directly from the map (Figure 22). An analyst can easily compare the number of days an ICAO is inactive, active, or exceeds constraints.

4.1.3 Drill-Downs

The drill-downs provided in this first visualization are a single tab book with three tabs (shown in the upper left corner of Figure 25). The first tab represents a detailed display of the

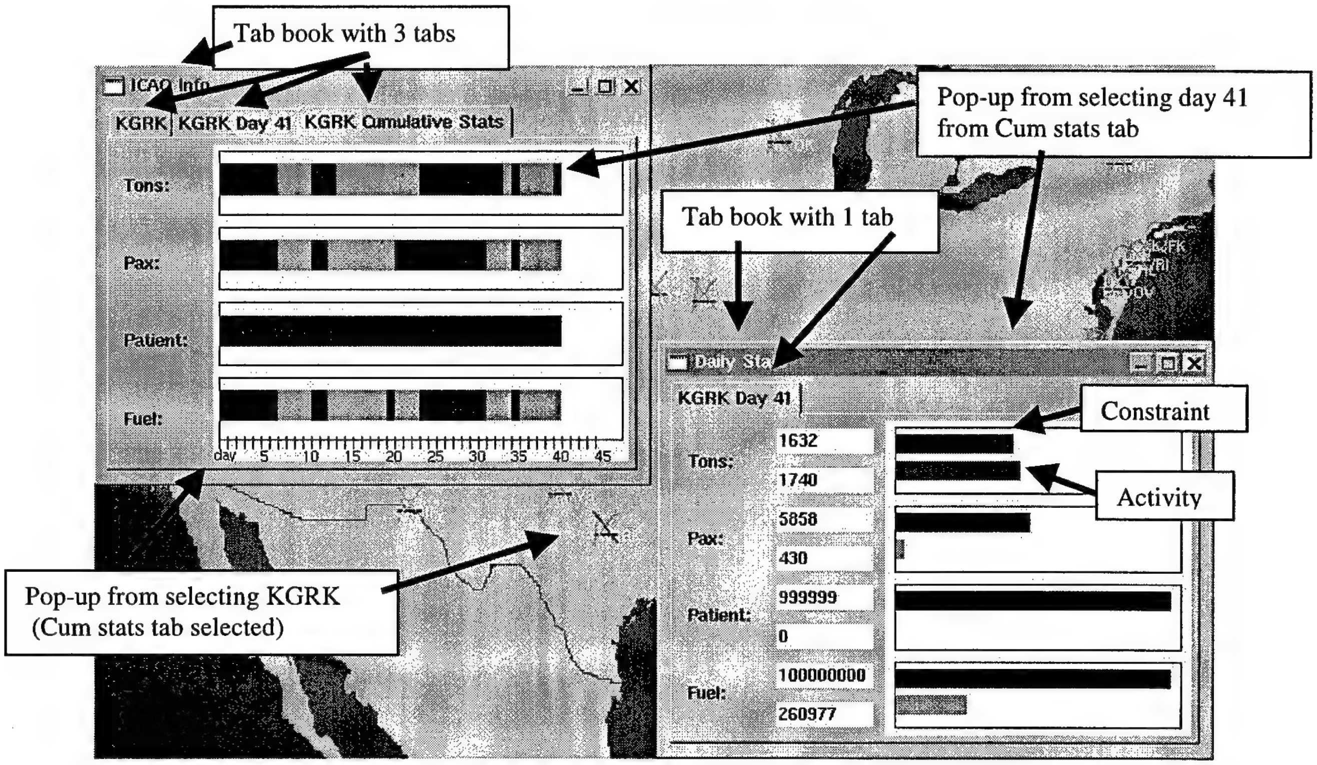


Figure 25: Main Pop-up Window and Day Selection Window (Plate 11)

location data for the selected airfield. The second tab displays a bar chart comparison of the constraints and activities of the selected airfield for the current day in each of the four categories. The final tab displays a bar chart comparison of the cumulative statistics of the selected airfield from day one to the current day (upper left Figure 25). The user can also select a specific day from the cumulative drill down by selecting the day on the bar itself. This selection of a day on the bar causes a pop-up window like the second tab of that day's specific statistics to display. Figure 25 shows the selection of the red status day in the left pop-up and the specific day information in the right pop-up.

4.2 Aircraft Tracking

The second application was chosen for its complex data structure requirements, its hierarchy of drill-down information, and its use of additional user interface devices. In AFM, aircraft tracking involves working with several different files and coordinating the data in these files. The aircraft and missions generated for the scenario are input files for the simulation. The current location of an aircraft is recorded in the leg information output file. By comparing the aircraft's leg information at a given time with the mission information from the mission information file, the aircraft's location and statistics can be determined. This application involves the development of more readers and a mix of hierarchical data structures. This application also highlights the correlation capability, not just the summary capability of the applications data structures.

This application shows the total volume of aircraft at a particular base at a given hour. The fact that the data in this application is hour-based and will exercise all of the hour-based user input devices was another reason for choosing it. The first application did not exercise these devices, so it complements the first application to give coverage of all the devices. This application also contains more integrated drill-downs than the first application, including lists, bar charts, and tables.

The components comprising this application and the abstract base classes they implement, interfaces they exercise, and other items exercised are shown in Table 6. For example, the `leg_infoReader` implements the `objReader` for the `inputMod`. It exercises the `addobject()` and `execute()` methods of the `objReader` class. It also is different from other readers in that it preprocesses the data as discussed below in the data structure section, rather than just reading data and putting it straight into the data object. Another example of a new item is the `msn_infoPlot` component, which implements the `objPlot` class, but it also is the only component to use the `inputMod's getData()` method with incremental reading.

Table 6: Component Descriptions for Application #2

Name	Type	Framework interfaces	Other items
loc_listReader	objReader	objReader - addObject(), execute()	floats, integers, char
loc_listDataMap	objData	objData - addobject()	"map" data structure
leg_infoReader	objReader	objReader - addObject(), execute()	data post procexxing
aircraftLocData	objData	objData - addobject()	maps and list data structures
msn_infoReader	objReader	objReader - addObject(), execute()	incremental reading
msn_infoData	objData	objData - addobject()	incremental array population
mapReader	objReader	objReader - addObject(), execute()	builds multiple data objs
waterData	objData	objData - addObject()	large array data structure
landData	objData	objData - addObject()	large array data structure
mapContext	objContext	objContext - all methods, inputMod - getData()	Display lists and large data sets for graphics engine
declutterVis	objVis	obj-Vis - Display(), DisplayLegend(), inputMod - getData()	Visual component update of data used by other components
aircraftLocVis	objVis	objVis - Display(), DisplayLabel(), DisplayLegend() inputMod-getData(), interfaceMod - getStatus()	single point display, view menu, one-hour increments, and object picking
aircraftICAOPlot	objPlot	objPlot - execute(), updatePick(), and inputMod - getData()	tabbook, and large OpenGL based selectable bar charts
tailsPlot	objPlot	objPlot - execute(), updatePick(), and inputMod - getData()	tabbook, and large OpenGL based selected text lists
leg_infoPlot	objPlot	objPlot - execute(), updatePick(), and inputMod - getData()	tabbook, Fox textfields
msn_infoPlot	objPlot	objPlot - execute(), updatePick(), and inputMod - getData()	tabbook, Fox textfields and exorcises incremental data reads
aircraftInterface	objInterface	objInterface - executeView()	Fox event loop, view menu
declutterInterface	objInterface	objInterface - executeAccess()	accessories menu additions

4.2.1 Data Structures

The leg information file has an entry for an aircraft each time it reaches an ICAO. This entry lists the aircraft's current leg information to include arrival and departure time. When an aircraft record is read, it is added to the ICAO's list of aircraft from arrival hour to departure hour. However, when an aircraft terminates a mission, it is shown as an entry at the ICAO with no departure time. If that aircraft gets a new mission, a whole new entry will show up at that same ICAO with the correct arrival time, but now with a departure time. This new aircraft entry supercedes the terminated mission entry. The problem is the terminated mission and the new mission are not linked in the file. This forces the `leg_infoReader` to take a second pass through the data checking for the terminated mission and new mission entry for the same aircraft. Without this check the terminated mission entry erroneously shows the aircraft parked at the ICAO for the remainder of the scenario.

The optimization of this two-pass problem and the displays needed for a total number of aircraft at a particular base per hour required an hour-based array at the top level of the data structure. Figure 26 is a depiction of this hierarchy where each hour (index of the array) contains a list of the airfields with aircraft, indexed by ICAO designator. Each airfield in the list

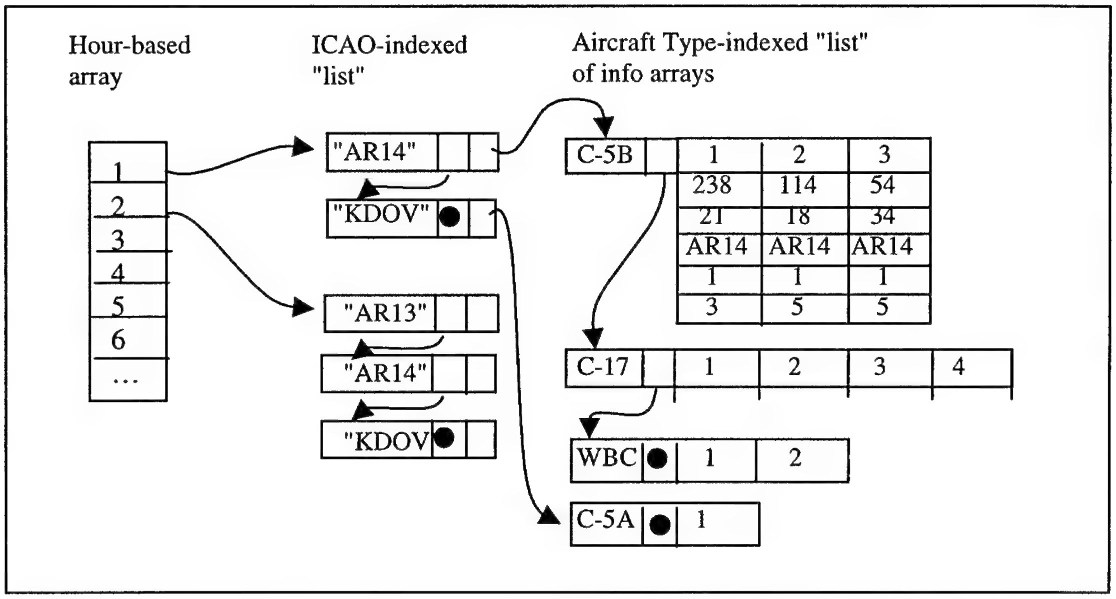


Figure 26: Application #2 Data Structure Diagram

represents a list of aircraft types at the airfield. Each aircraft-type list entry is an array of the individual aircraft information. This hierarchy allows the display of aircraft quantities to step through the array getting the list size for each airfield. When the quantities of the different types are needed for the micro view, the size of the aircraft type array is all that is needed. The hierarchy and easy size calculations keeps searches and passes through the data structure to a minimum, because the most frequently needed data is available at the top of the hierarchy.

By using an array at the top level and using a second temporary list of mission-terminated aircraft, the leg_infoReader component can optimize its overall execution time. The leg_infoReader's temporary list of terminated aircraft can be used to reduce the second pass through the data to specific indexed checks at those hours and ICAOs with terminated aircraft instead of a complete pass. The checker portion of the leg_infoReader merely picks the first aircraft off of the temporary list and looks at the ICAO's list until it finds another instance of that aircraft with a departure time. If another is found it deletes the entry from the temporary list knowing the new mission superceded it. However, if another entry is not found, the terminated mission must be inserted at the ICAO from the arrival time to the end of the scenario. This enables the checker to jump in at the point of a terminated aircraft and not repeatedly look for conflicts at all ICAOs for all hours. The trade off of maintaining the additional lists far out weighed the initial slow execution time of leg_infoReader.

Another optimization of the leg_infoReader is with the implementations used for the lists. The mixed data structure and use of a temporary list improved the readers performance over the initial solution, but the execute() method was still taking several seconds due to the size of the data files. The first implementation of this array and list mixture used the C++ STL "list" classes for the lists, but the performance of the STL was very slow. The final implementation of this data structure uses purpose-built linked lists. This change decreased the execute time for the leg_infoReader by almost 60% for a 90-day scenario.

The msn_infoReader reads the mission information file and populates a single array of mission data. The missions for a scenario are indexed with a unique mission number. Research of the data revealed that the list of mission numbers has missing numbers, but the density of the numbers is sufficient to make an array the most effective data structure. Each array entry is a record of the mission information including the onload and offload airfield, cargo sizes and weight, and arrival and departure information. This simple array is also justified by the need for direct access to the mission information.

4.2.2 Visual Display

This application has both a micro and macro view of the aircraft data. Both of these views can be used to "playback" the MBs of data on an hour-to-hour basis from the same window and scene. The macro view (Figure 27) of the data shows a single bar whose height represents the total quantity of aircraft at an ICAO at that hour. This macro level view enables the analyst to

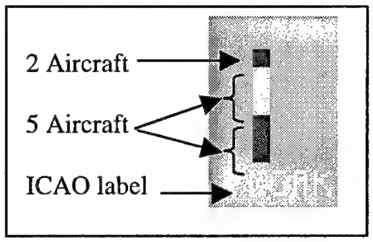


Figure 27: App #2 Macro Symbol (Plate 8)

directly compare the quantities of aircraft at the different ICAOs directly from the map (Figure 28). To aid in the direct comparison at the macro level the bar is striped with five aircraft per colored stripe. This view alone provides the analyst a display of MBs of data in a single scene.

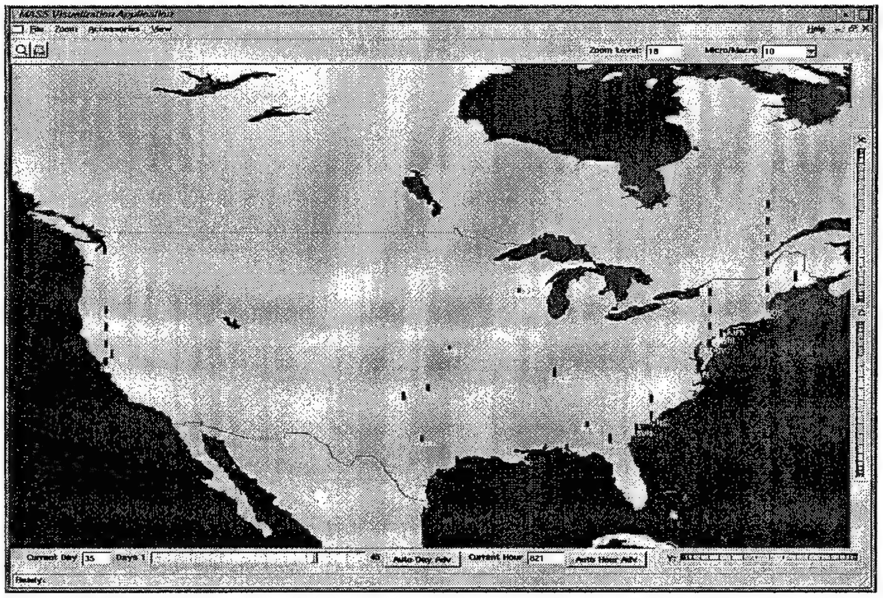


Figure 28: Application #2 Macro View (Plate 12)

The micro view uses a spider-like glyph showing the quantities of aircraft by aircraft type at the ICAO (Figure 29). The AFM simulation can support up to fifteen different types of aircraft in a scenario, so this glyph is designed with that in mind. Each color-coded leg of the spider

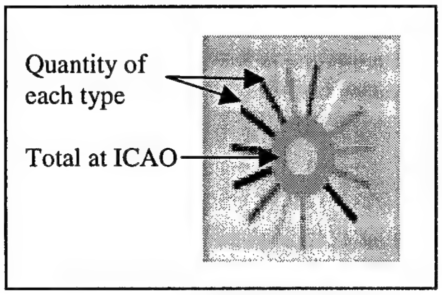


Figure 29: App #2 Micro Symbol (Plate 9)

represents a different type of aircraft as listed in the aircraft types file of the AFM input. The quantity of each type of aircraft is reflected in the presence and length of each leg. Figure 29 shows the test case of a quantity of 20 aircraft of all fifteen types of aircraft. Additionally the fullness of the green center body of the spider represents the total number of aircraft at the ICAO. This micro view enables the analyst to compare airfield aircraft quantities by aircraft type directly from the context map (Figure 30). The fullness of the spider body provides total quantity

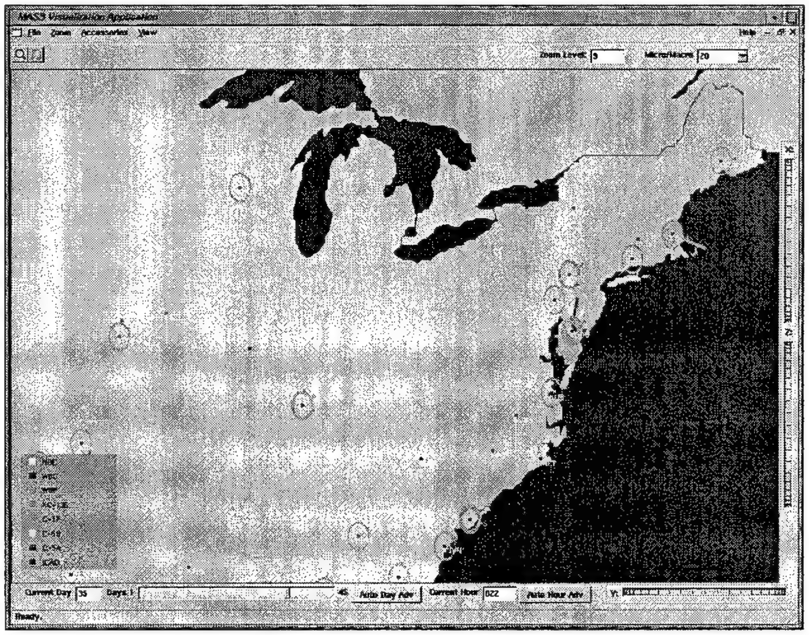


*Figure 30: App #2 Micro View (*Plate 13*)*

comparison like the bar length of the macro view. The spider-like legs provide type-quantity comparisons at the global level without drill-down.

In addition to these aircraft tracking displays, this second application also has a `declutterVis` component that is added to the "Accessories" display. The left-hand picture of Figure 31 shows a regular view of the aircraft data in the Northeastern US. The right-hand

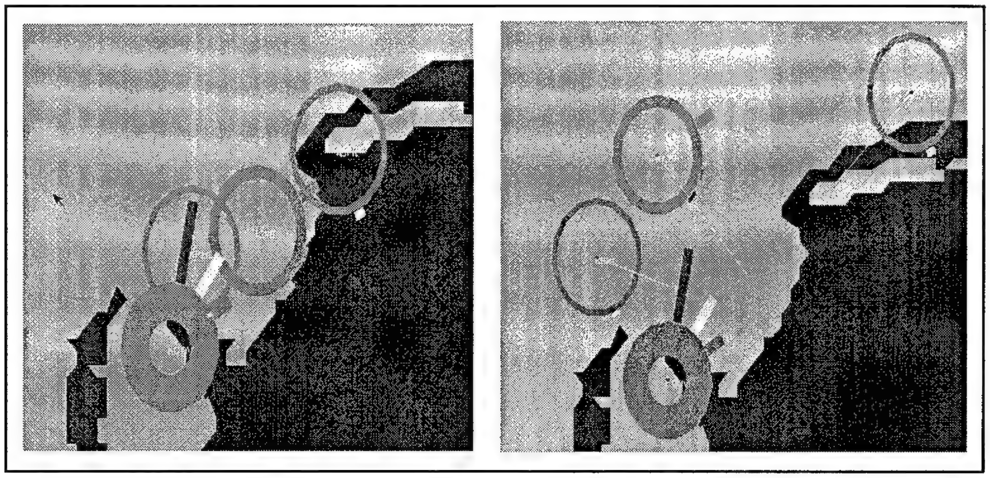
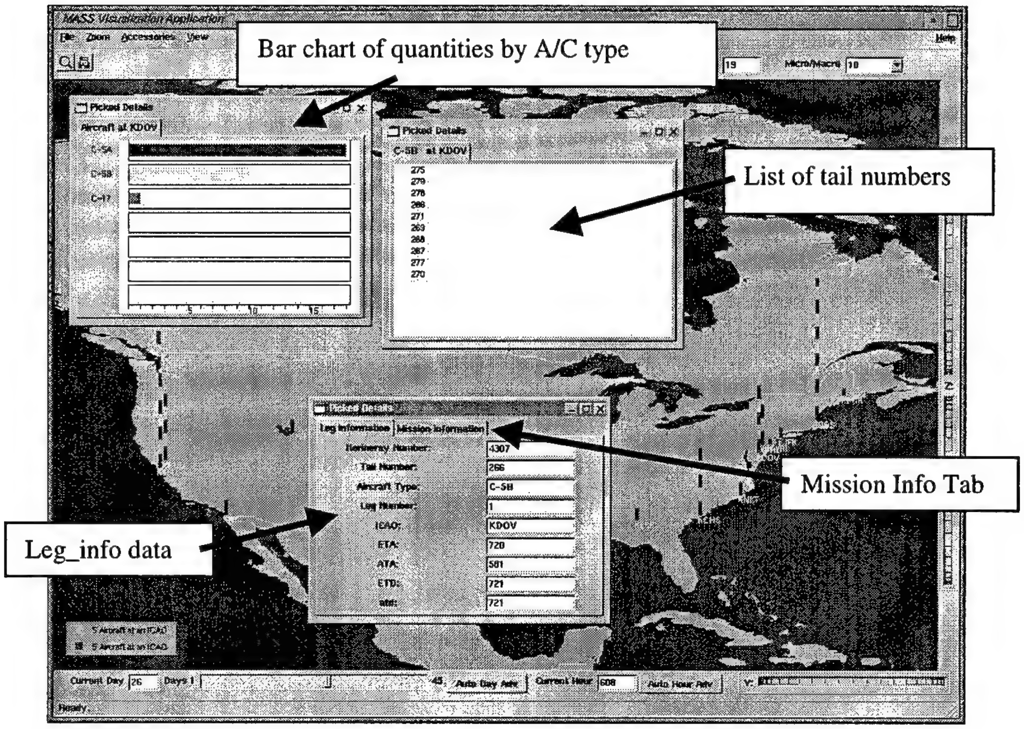


Figure 31: Application #2 Declutter Example (Plate 14)

picture shows this same data with the declutter function turned on. The declutter function replaces the ICAO's location with new coordinates and renders a line pointing back to the original location. This enables the viewer to get a clearer picture of the glyphs. The original ICAO location is saved and replaced when the declutter function is deactivated.

4.2.3 Drill-Downs

The drill-downs start with the user selecting an ICAO's bar from the micro or macro level. The drill-downs for this particular application provide several levels of analysis for the user. Figure 32 shows the three drill-down windows. Selecting a striped bar from the context view pops-up a window with a bar chart listing the quantity of aircraft by type for that ICAO (upper left Figure 32). The user can further drill-down by selecting an aircraft type, so a pop-up



Itinerary Number: 4307
Tail Number: 266
Aircraft Type: C-5B
Leg Number: 1
ICAO: KDOV
ETA: 720
ATA: 581
ETD: 721
atd: 721

Figure 32: App #2 Drill-down Windows (Plate 15)

window will appear with a list of tail numbers of the aircraft comprising that type at the ICAO (upper right Figure 32). These tail numbers can be explored by drilling-down on a tail number. Selecting a tail number will show the current leg information and mission information for the aircraft at that time (lower pop-up Figure 32).

4.3 Implementation Summary

The daily airfield statistics and aircraft tracking applications were both able to provide the coverage needed to meet the criteria outlined in Chapter 3 of this research. Table 3 on the next page shows the mapping of the components implemented in these two applications to the criteria discussed and listed in the methodology section of this AFM visualization research. Each application exercised the framework for objective #1, which was to develop a robust information visualization architecture. They also validated the designs for the components for objective #2, which was to develop reusable components for future visualization applications. Both applications successfully produced visual representations for improved analysis for objective #3, which was to implement visualization applications to improve AFM analysis capabilities. In addition, they both implemented different levels and types of data structures, visual depictions, drill-downs, and user interactions. As shown by the coverage of the framework success criteria listed in Table 7, the two applications were able to validate the framework and produce the results discussed in the next chapter.

Table 7: Component to Coverage Criteria Mapping

Obj	**Type/ Area**	**Criteria**	**App #1**	**App #2**
1	Fr. Interface	objReader -- addObject(), execute	✓	✓
1	Fr. Interface	objData -- addObject(), fully populate	✓	✓
1	Fr. Interface	objData -- addObject(), incremental populate		✓
1	Fr. Interface	objContext -- addObject(), all interfaces	✓	✓
1	Fr. Interface	objVis -- addObject(), Display(), DisplayLabel(), DisplayLegend()	✓	✓
1	Fr. Interface	objPlot -- addObject(), updatePick(), execute()	✓	✓
1	Fr. Interface	objInterface -- addObject(), executeView()	✓	✓
1	Fr. Interface	objInterface -- addObject(), executeAccess()		✓
1	Fr. Interface	inputMod -- getData()	✓	✓
1	Fr. Interface	visualMod -- Callbacks and execute()	✓	✓
1	Fr. Interface	plotMod -- CallbackPicked()	✓	✓
1	Fr. interface	interfaceMod -- getStatus()	✓	✓
1	Data	Array data objects	✓	✓
1	Data	C++ STL based data objects	✓	✓
1	Data	list data objects		✓
1	Data	Hierarchical data objects		✓
1	Visual	Micro/Macro view	✓	✓
1	Plot	Multi-tab tab book	✓	
1	Plot	Pick items on drill-down window	✓	✓
1/2	Plot	Bar chart drill-down window	✓	✓
1/2	Plot	Text fields and data values	✓	✓
1/2	Plot	Text lists		✓
1/2	User Interface	View menu additions	✓	✓
1/2	User Interface	Access menu additions		✓
1/3	User Interface	Day button and slider	✓	
1/3	User Interface	Hour button and slider		✓
1	User Interface	Working Window	✓	✓
3	Visual	Global summary of data set	✓	✓
3	Visual	Context global comparison of data from the data set	✓	✓
3	Visual	"Playback" of scenario day by day or hour by hour	✓	✓
3	Visual	Data processing or enhancement for better analysis		✓

5 Results

This chapter examines the AFM visualization research results and analyzes them with respect to the stated objectives. The first section discusses the achievement of the objectives by reviewing the objective and discussing the success of the research in meeting that objective. This chapter discusses the specific validations of the framework, the design and code reuse levels, and improved analysis capability provided by the two sample applications. The chapter concludes with a summary of the results.

5.1 Objective Achievement

A measure of research success is the degree to which it satisfies the objectives and its goals. In this AFM visualization research the primary objectives were to develop a component-based architecture that supported reusable components. The use of these two objectives in implementing application components was an integral part of satisfying the third objective of producing an improved AFM analysis tool in the form of a successful visualization of the simulation's output.

5.1.1 Architectural Framework

This first objective was to develop a robust information visualization architecture that supported a component-based system implementation. To achieve a component-based architecture for the system a stable underlying "engine" or framework was needed. This framework provides the common functionality and defines the interfaces for the components. This AFM visualization research designed and implemented a four-module application framework, consisting of input, visual, plot and interface modules. This architecture was tested and validated with the development of two visualization applications.

The architecture was initially validated by the fact that these two sample applications could successfully operate on the same four modules and interfaces. These two applications represented diverse aspects of the AFM data and the compliment of the two applications covered all aspects of success criteria outlined in the methodology section of this thesis and listed here in Table 8. The specific component to criteria mapping is provided in the end of Section 4.3 of this research.

Table 8: Framework Robustness Success Criteria

Type/Name	**Description**	**Success Criteria**
Framework Interfaces	(Internal) Layer to layer communication of the framework	The framework can manage the applications and execution with the interfaces provided.
	(External) Component to framework communication	Application can compile and execute with components only using defined framework interfaces.
Data	Data structures used in Data Objects	Diverse data sets can be handled transparently by the framework and components
	Population of data structures	Data objects can be populated in whole or incrementally by the framework and applications
	Reading of data and preprocessing	Data can be read in directly or preprocessed transparently by the framework and application.
Visual	Micro and Macro views	System can dynamically provide macro and micro levels of detail when the user selects it.
	Context based displays	Complete data set viewing with a single context
	Pick items on the display	User can select display items for drill-down
	Animation or time based display	Display can handle changing of time of simulation
Plot	Single and Multi-tab books	Display of single and multi-tab pop-up windows
	Pick items on pop-up windows	User can select display items for drill-down
	Diverse drill-down windows	The framework and applications can handle different formats display in the drill-down windows
User Interface	Menu additions with components	The framework can handle menu additions to the "Accessories" and "View" menus.
	Time controls (sliders and buttons)	Framework can handle the different user interface devices that change the current scenario time.
	Working window and status display	Framework displays working window and "working" on status bar when doing lengthy background work.

The framework design enabled the two applications to be implemented with 36 independent components. Through the course of developing the 36 different components, the interfaces provided by the framework proved to be complete and made direct component-to-component communication unnecessary. All uses of a component by another component are done exclusively through the interfaces of the framework. This further satisfied the component-based design portion of the objective, because independent components means that they can easily be mixed and matched as well as included or excluded from the system.

This independence of components and robust application framework were further validated by including and excluding different components from the two applications. The exclusion of a particular component tested the other components' independence at two levels. The first level was the compilation of the executable itself and the second was running the application.

Table 9 and Table 10 show a list of the components that make up each application and which components were excluded for each test and the outcome. For example, the first test

Table 9: Visualization #1 Inclusion and Exclusion Test results

Visualization #1 components:			
Readers:	**Visualizations:**	**Plots:**	**Interfaces:**
scenarioReader	baseVis	basePlot	baseInterface
mapReader	airfieldStatVis	statsPlot	statsInterface
loc_listReader	airfieldCumVis	cumPlot	cumInterface
summaryReader			

#	Excluded	Result
1	scenarioReader mapReader loc_listReader summaryReader	Main window came up correctly, but blank. No visualizations were possible, because no data was present. Error messages for each data not found as the visualizations were picked.
2	mapReader	Everything worked except the map, display was blank blue main window.
3	scenarioReader	error message: "no scenario data, to find input directories and sizes"
4	loc_listReader	No visuals display, they are all based on the loc_list data, map still worked correctly.
5	baseVis	No affect, when selecting on "ICAO" menu item, nothing is drawn
6	airfieldStatVis	Like baseVis, selecting menu, nothing rendered. Cumulative stats render correctly.
7	AirfieldStatVis airfieldCumVis	Neither daily statistics nor cumulative statistics render when their menu items are selected. No error messages, worked as expected.
8	basePlot	BasePlot's tab not present in tab book of an ICAO selection. Everything else works
9	basePlot statsPlot cumPlot	When and ICAO is selected, the error message "no ICAO plots found" appears and no drill-down window is rendered. Everything else works as expected.
10	baseInterface statsInterface cumInterface	None of the interfaces can be selected, because the menu items are not present on the menu. Everything else worked as expected.

Table 10: Visualization #2 Inclusion and Exclusion Test results

Visualization #2 components:

Readers:	**Visualizations:**	**Plots:**	**Interfaces:**
scenarioReader mapReader loc_listReader leg_infoReader men_infoReader	aircraftLocVis	aircraftAtICAOPlot tailsPlot leg_infoPlot msn_infoPlot	aircraftInterface

#	Excluded	Result
1	scenarioReader mapReader loc_listReader leg_infoReader msn_infoReader	Main window came up correctly, but blank. No visualizations were possible, because no data was present. Error messages for each data not found as the visualizations were picked.
2	mapReader	Everything worked except the map. Everything displayed on a blank blue main window.
3	scenarioReader	error message: "no scenario data, to find input directories and sizes"
4	leg_infoReader	
5	msn_infoReader	The msn_plot data tab was not included in drill-down, because msn_info data could not be found. Everything else worked as expected.
6	aircraftLocVis	Nothing is drawn or even read in when the menu item is selected
7	aircraftAtICAOPlot	No drill-down window comes up when you pick on an ICAO.
8	tailsPlot leg_infoPlot msn_infoPlot	Drill-down window for ICAO comes up, but there is no response to further drill-downs on that window, because no component is registered for those handles.
9	Leg_infoPlot	When clicking on a tail number, nothing happens as expected.
10	aircraftInterface	Aircraft visualization could not be selected. Everything else worked.

excluded all of the reader components required by the application. Compiling and producing an executable without the components means the included components are truly independent of these readers at that level (none of them "#include" any of these). Running the application with these components missing resulted in the main window and interfaces coming up correctly, however, there was no map context present in the window. Any interaction by the user resulting in components requesting data resulted in error messages being produced by the input module reflecting the fact that the data was not available. Additional tests isolated the various other components to test the framework and application's ability to handle the absence.

All of the tests shown in Table 9 and Table 10 were successful from two standpoints. First, the components were again found to be independent and easily included or excluded by the developer in the `starter` file. Secondly, the framework handled the different configurations by not exiting prematurely. As expected the resulting system behavior is not what the user would want visually, but the system handled the events gracefully and did not crash.

The user interface is another aspect of the application framework that was successfully implemented in this research effort. Information visualizations are used by a wide array of users, so the interface must be intuitive and yet robust. The interface in this research provides that effective interface for the analyst looking for summaries and playbacks. The interface is straightforward, gives a majority of the real estate to the task at hand in the display window, and all devices have hints that display in the status bar. At the same time, the user interface is robust enough to support the serious analyst wanting details at all levels.

The architectural objective of providing a common framework that supported the development of independent component-based applications was met. The robustness of the architecture is sufficient to support the AFM visualizations. The interfaces were also sufficient to support the demands of the requirements of AFM and other visualization applications.

5.1.2 Reuse

The second objective of this research was to develop components that support reuse for future component development. In object oriented programming the number of reused classes is a common measure of reuse, but in this research every component is a complete class, so the reuse of a complete class would be almost zero. So, this research looked at design and specifically code reuse levels.

Although classes as a whole were not reusable in this research, the design of each class was reused for each subsequent component of the same type. For example, visual components all providing implementations for the `objVis` abstract base class have the same design. This reuse of design ultimately led to the high source code reuse numbers found in this AFM visualization research.

Source code reuse is calculated by determining how much code from the first application was reused in the second application. Table 11 shows the amount of reuse from the first application to the second application. Each object in the second application is listed with the lines of code for the object, followed by the reused lines of code, and finally the percentage of reuse for the object. The average reuse percentage for the components is 84.4%. The total overall percentage of reuse from the first application to the second application was 83.8%, much higher than originally anticipated.

Table 11: Reuse Statistics for the 2nd Visualization Objects

Object Name	**LOC**	**Reused LOC**	**%**
leg_InfoReader	424	308	73%
aircraftLocData	103	95	92%
aircraftICAOMap	176	166	94%
aircraftAtICAO	192	162	84%
leg_Info	114	102	89%
msn_infoReader	492	420	85%
msn_infoData	102	99	97%
msn_info	174	104	60%
aircraftLocVis	415	355	85%
aircraftICAOPlot	793	738	93%
tailsPlot	569	509	89%
leg_infoPlot	456	293	64%
msn_infoPlot	563	467	83%
aircraftInterface	160	150	94%
Total:	4,733	3,968	84%

One note on the exceptionally high level of reuse is the similarities between the visualizations of the AFM output data. Initial review of the data revealed diversity in the AFM data, but in the design and implementation of the different applications it was discovered that most of the components are similar from the aspect of design and execution, just not necessarily data types. Because the design is so generic and yet affective, the percentage of code used for the unique graphics rendering and data structure is small compared with the stable framework interface code. As a result, this high amount of reuse is not unreasonable for visualization of the AFM output,. The design will yield comparable results even with a system other than AFM. This high level provides further support for showing the completeness of the framework and its ability to support the visualization of data.

Another tangible byproduct reinforcing reuse is the difference in development time for the two visualizations. The 11 components of the first visualization required approximately 50 hours to implement. The second visualization required approximately 20 hours. With 14 classes in the second visualization, the average development time was 1.42 hrs/object compared with 4.54 hrs/object in the first visualization.

The high level of reuse and the large improvement in development time both show measurable results of satisfying the second objective; which is using components that support high reuse levels for future development. The structure of the framework supports the use of independent components and the framework's defined interfaces make the design as well as the source code of those components reusable. The levels of reuse are projected to be high, because the common design-patterns between the components enables reuse of all but the specific data and graphics portions of the components.

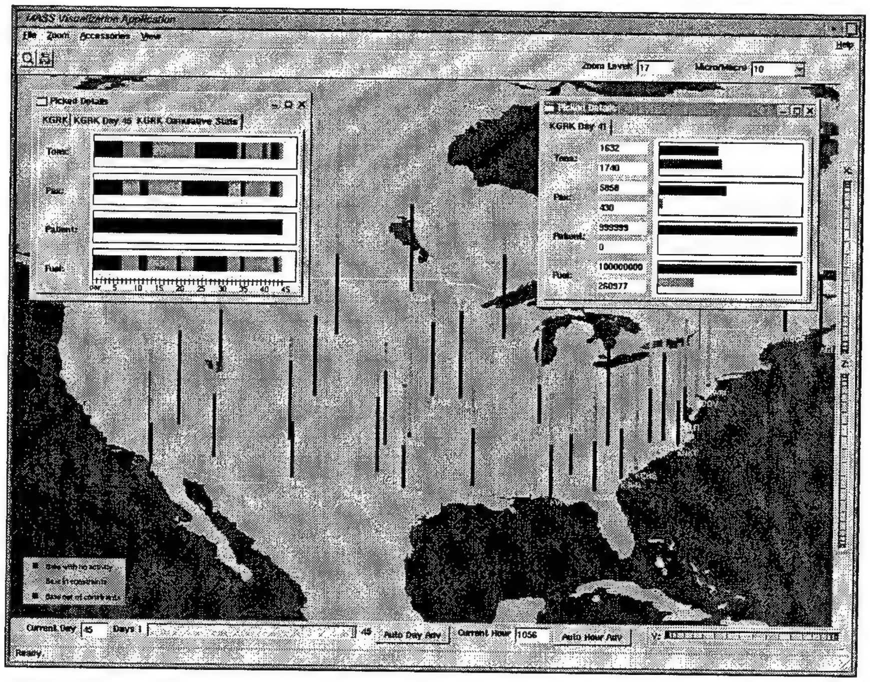
5.1.3 Implemented Applications

The third objective of this research was to develop and implement AFM visualization applications that improved analysis capability by addressing those problems of data context, drill-down, correlation, and integrated views mentioned by Peterson in *The Visually Enabled Enterprise: Managing Information Through The Power of Visualization* [17]. To show that this visualization system could provide these capabilities missing in previous analysis tools, two distinct applications that visualized AFM output data were implemented. These two visualizations were chosen for their representation of the different types of data present in the AFM output and their coverage of the framework's success criteria. This section looks at the success of these two applications.

5.1.3.1 ICAO Daily Statistics

The first implementation displayed the individual ICAO's daily constraints and activities in the areas of short tons of cargo, patients, pax, and fuel. Constraints represent the limit on the amount of that item that an ICAO can handle in a day. Activities represent the amount of that item that the ICAO received for that day. Using the display of these daily values the analyst can look at a particular day's worth of data and immediately compare it to all of the ICAOs across the globe for that same day. User control of the micro/macro level cutoff enables the analyst to get a summary of the ICAO as a whole or as the four individual parts of cargo, pax, patients, and fuel. This complete global view of a day's worth of data coupled with drill-down capability is magnitudes faster and more direct than previous efforts to analyze the same data. Previous efforts involved exporting the data to a spreadsheet tool, manually stripping off the extra data fields in the data file, selecting the right fields to draw a bar chart of a single day's activity and manually correlating the constraints with activities. Drill-down and geographical comparison capabilities were non-existent. With this research's first application, a day's activities are one click away and drill-down information is a second click away.

Beyond this new daily comparison capability, the first application has a cumulative display of the same data. With this cumulative display the analyst can perform trend analysis and instant comparison of all ICAOs across the globe for the entire simulation run. Figure 33 shows how the global view with drill-down windows can aid the analyst. The background of Figure 33 shows the cumulative bars as they appear in the main context display. These bars group like status together so the ICAOs can be compared directly. To see the actual order in which the status occurred, the user selects one of the ICAOs. A drill-down window like the left-hand one in Figure 33 pops up containing the status ordered by day. If an analyst wants the details of a single day, they right-click on that day's bar and the window on the right of Figure 33 comes up, showing the specific day's numbers.



Zoom Level: 17
Micro/Macro: 10

KGRK Day 41

Tons:	1632
	1740
Pax:	5858
	430
Patient:	999999
	0
Fuel:	100000000
	260977

Current Day: 45 Days: 1 45 Current Hour: 1056

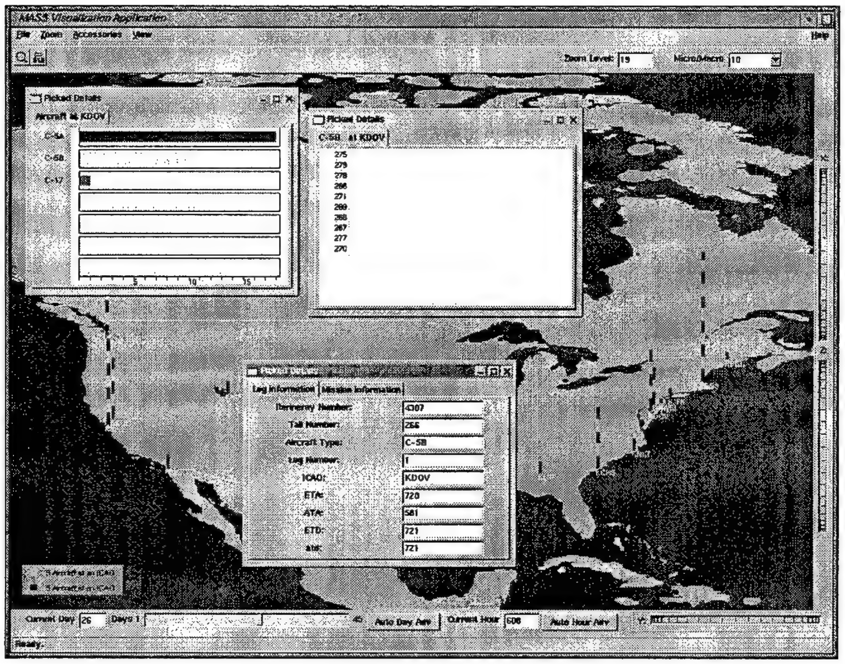
Figure 33: Application #1 Screen Capture (Plate 16)

Based on the limited spreadsheet capability of the previous analysis efforts for AFM, the cumulative global comparison is impossible. Previous efforts have written small programs to parse the data and display charts of specific information, but these programs were limited in scope. This cumulative display provides the analyst never before seen capability to summarize and playback entire MBs of scenario data in a simple integrated manner.

5.1.3.2 Aircraft Tracking

The second application, aircraft tracking, was chosen for its complex data structures and its additional coverage of the framework success criteria. The aircraft tracking application is a graphical depiction of the correlation of AFM's leg and mission information files. With this aircraft tracking application an analyst can see the quantities of aircraft at an ICAO for a particular hour or for the entire scenario by incrementing time. Utilizing the automatic increment capability for scenario time provides an analyst a virtual playback of the aircraft movement around the globe. While viewing this summary and playback, the analyst is only a mouse click away from detailed aircraft information. This playback correlation, and link to drill-down capability is unprecedented in AFM analyst efforts. Previous efforts have provided the ability to sort the leg information file and group all of the entries by tail number. This grouping was then used to manually step through the entries of the aircraft to analyze the mission items. No geographical or time correlation was possible in these groupings.

The drill-down capability of the aircraft tracking application alone is beyond any previous analysis capability for the AFM, because there was no automatic link between the leg and mission information files. Now with this application the bars of aircraft totals at an ICAO can be selected to show the quantities of aircraft by type. The type of aircraft can then be picked to see the tail numbers of the aircraft making up that quantity. From there a tail number can be selected to show the aircraft's specific leg and mission information. Figure 34 shows several of these drill-down windows. The left-hand drill-down is a count of aircraft at "KDOV" ICAO by type. The upper right-hand window in Figure 34 is a list of C-5B tail numbers that were requested by clicking on the C-5B bar in the first window. The bottom window in Figure 34 is the detailed information on tail number 266. This window also contains mission information on the other tab.



Zoom Level: 19
Micro/Macro: 10

275
279
278
266
271
269
268
267
277
270

Itinerary Number: 4307
Tail Number: 266
Aircraft Type: C-5B
Leg Number: 1
ICAO: KDOV
ETA: 720
ATA: 581
ETD: 721
atd: 721

Current Day 26
Current Hour 606

Figure 34: Application #2 Screen Capture (Plate 17)

5.2 Results Summary

This AFM visualization research achieves and exceeds all of its objectives. These results were achieved with integrated data structures and by leveraging good software engineering of the application framework. The application framework and its interfaces are robust, reliable and consistent. Significant reuse of component design and code greatly reduces development efforts. The applications developed in this research are unparalleled by any previous AFM analysis techniques. These applications provide the analyst with clear and concise summaries and playbacks of tens of thousands of data records in a simple intuitive user interface. At the same time the applications' drill-down capability provides a means for the analyst to delve deeper into the details of the data.

6 Conclusions and Future Work

6.1 Conclusion

Air Mobility Command's (AMC) Airlift Flow Model (AFM) is a modern simulation that takes advantage of advancing computer technology. This simulation runs multi-day mobility scenarios in a matter of minutes producing megabytes of output data. Because the analysts' needs for summaries, trends, and comparisons of the data have surpassed the capabilities of desktop spreadsheet routines, new tools are needed. Information Visualization is one of these new tools. Liberating the information contained in the large volumes of data and visualizing it was the central goal of this AFM research. The result was a robust component-based visualization application framework for AMC.

This AFM visualization research required the creation of a robust application framework and several component-based data visualizations utilizing the framework. The four-part application framework consists of an input, visual, plot, and interface module and successfully provides the reuse, flexibility, and component support needed to meet the AFM visualization research objectives. These four modules handle all the management and execution of the system's component-based implementations. The object-oriented nature and thorough design of the framework, interfaces, and components give the research effort higher than expected levels of reuse. With the stable visualization engine provided by this framework the application developer can concentrate on the task at hand and not deal with the underpinnings of the system.

Two component-based visualization applications were implemented for this research. These two representative applications show that information visualizations can help with those concerns of data context, drill-down, correlation, and integrated views mentioned by Peterson in *The Visually Enabled Enterprise: Managing Information Through The Power of*

Visualization[17]. Both displays provide integrated views and global comparison capability in a single display with a map context. Drill-down and micro/macro levels of detail are also present. These two visualizations are integrated in the same display, so an analyst can view the different data together in the same context. The straightforward user interface enables the analyst to manipulate the data for their needs.

These capabilities contribute to better analysis tools for the AFM simulation model. As a result of this research, the AFM analyst has a powerful analysis tool that can be easily extended to meet additional needs. Demonstrations already conducted with AMC personnel have liberated parts of the AFM output data that had never been directly analyzed. This capability will provide AFM analysts and system designers a way to answer our leaders' mobility questions for years to come.

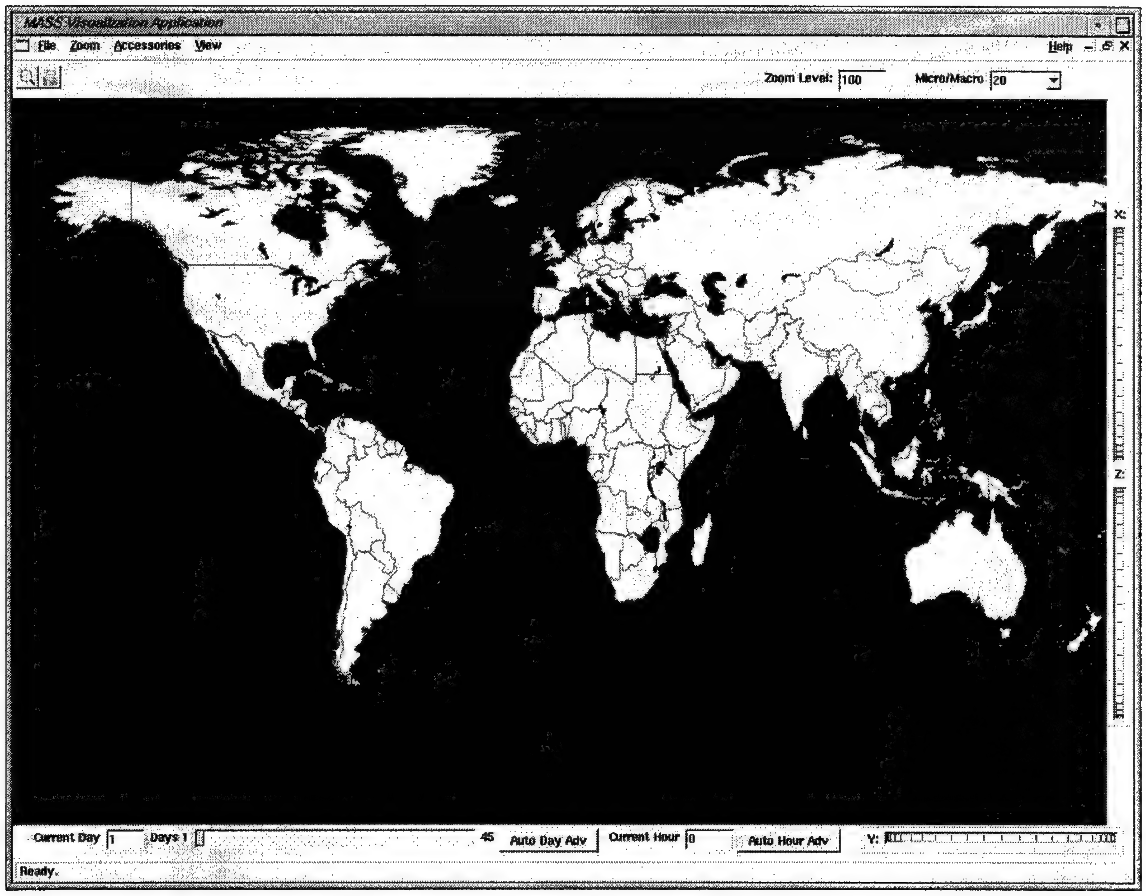
6.2 Future Work

There are three areas where future work can be completed with this information visualization framework. The first is to enhance the framework to support more dynamic user interface widgets. The ability to add other interface widgets like dials and buttons with interface components would improve the robustness of the application framework. Future applications of this framework in areas other than AFM will require this dynamic capability.

The second area for future work is using the framework with another application or domain. The AFM simulation analysis was the main requirement for this research, but the framework was designed to support the needs of all information visualizations not just AFM. The validation of the framework and the reuse results in this research show that the framework is a good solution for any visualization effort. The interfaces and design are generic enough to support other applications. Exploring other applications would be beneficial follow on research.

The third area of future work is with additional applications for AFM analysis. A large portion of this effort was spent on the design and development of the underlying framework. Two representative applications were developed to visualize portions of the AFM output, but there is a great deal more to visually represent. Additionally, AFM data with different contexts exist. Data such as crew information, weather, and aircraft parking patterns need to be visualized with contexts other than a map. Developing visualization applications for this portion of AFM data is needed by AMC. With the application framework completed, this work would deal more with the actual visualization of the data.

Color Plates



Zoom Level: 100
Micro/Macro 20
Current Day 1
Days 1 45
Current Hour 0
Ready.

Plate 1: Screen Capture of Main Window with Map Context

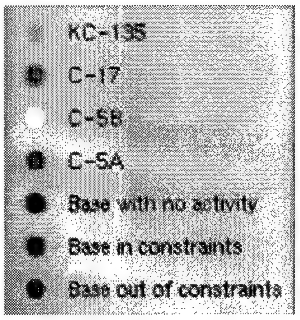


Plate 2: Picture of Legend

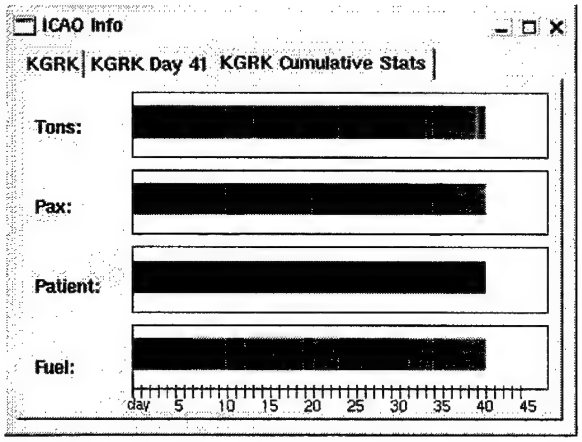
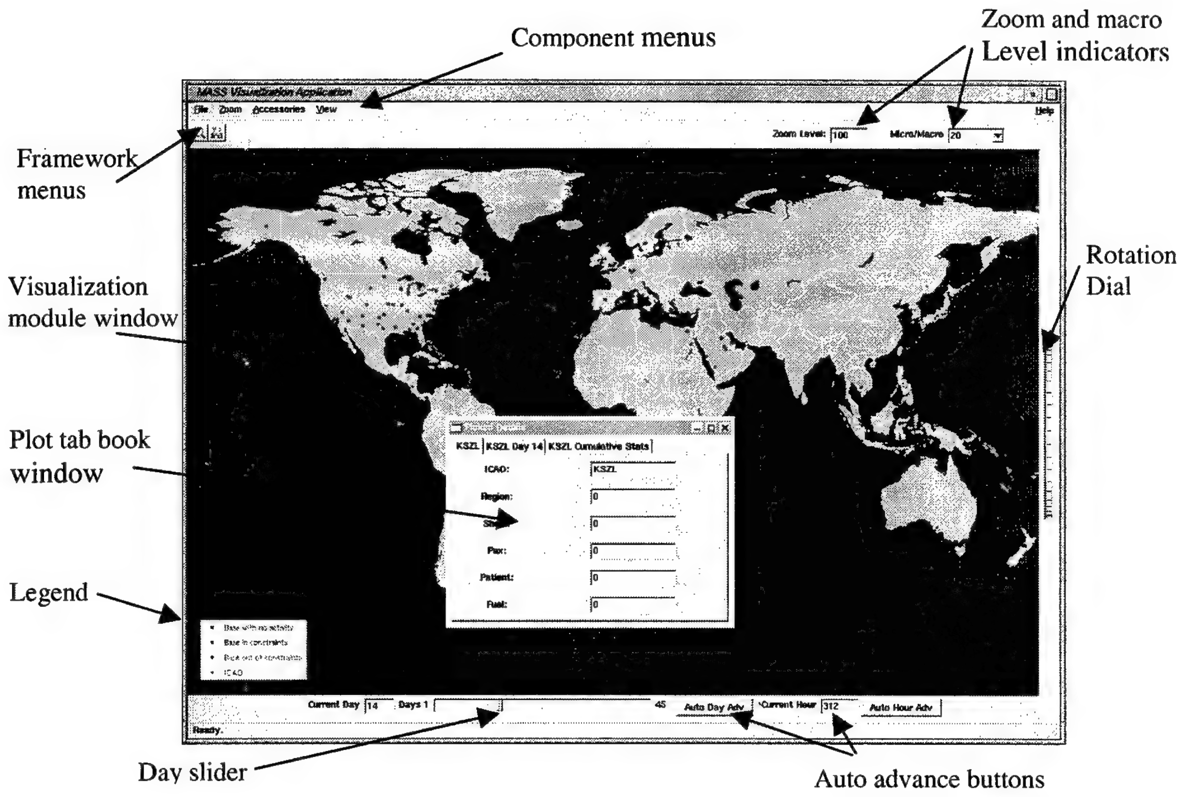


Plate 3: Picture of Plot Module Tab Book



ICAO:	KSZL
Region:	0
Str:	0
Pax:	0
Patient:	0
Fuel:	0

Plate 4: Picture of User Interface

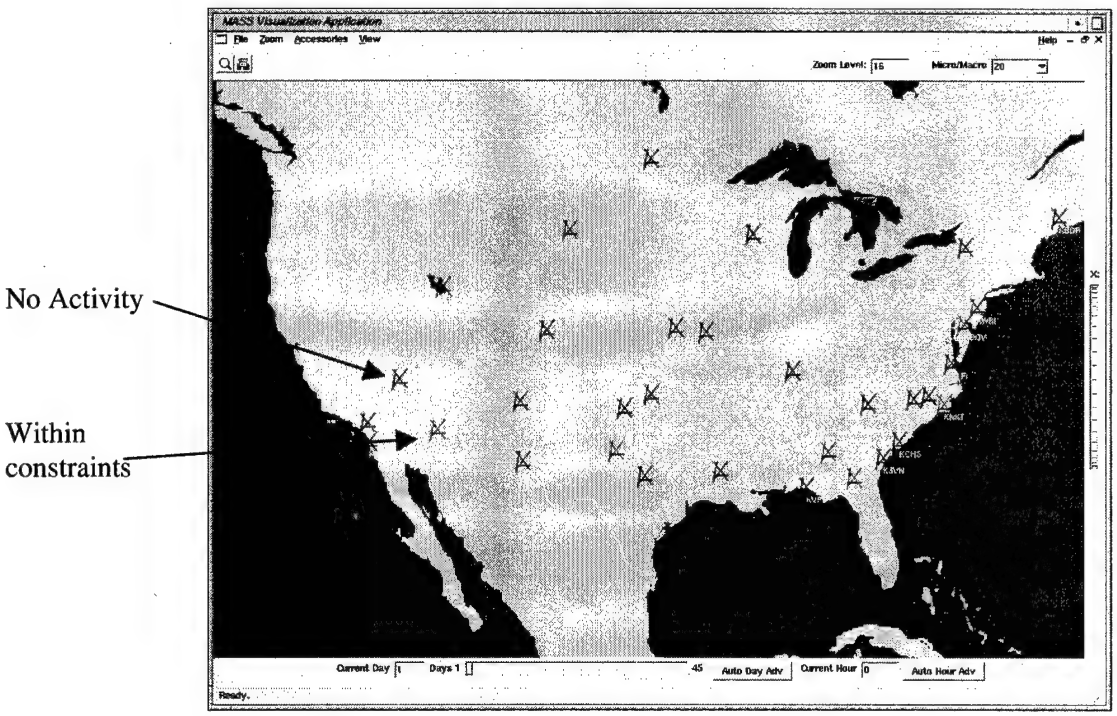


Plate 5: Application #1 Micro View

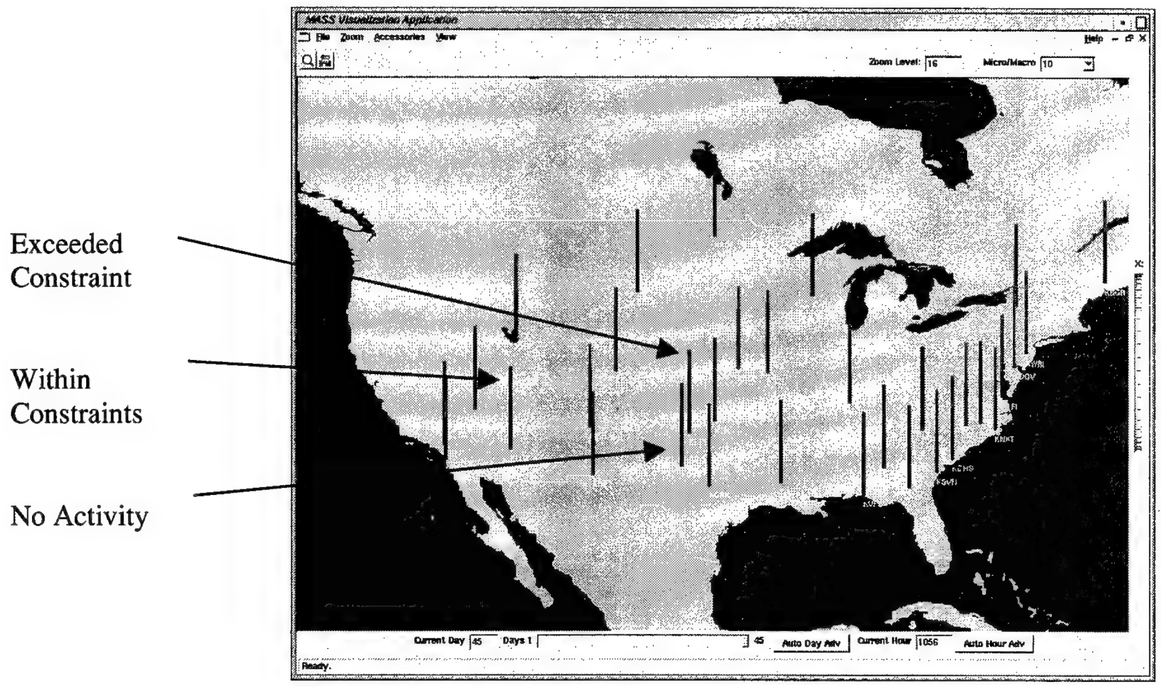


Plate 6: Visualization #1 Cumulative View

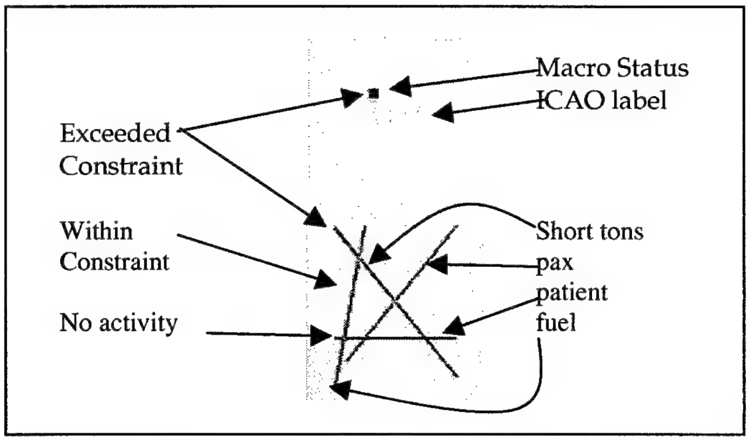


Plate 7: Statistics Symbols

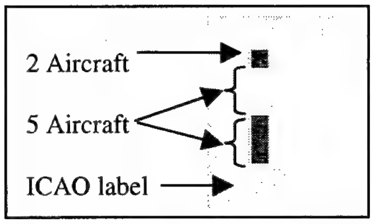


Plate 8: App#2 Macro Symbol

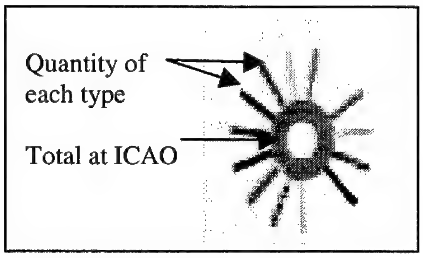


Plate 9: App #2 Micro Symbol

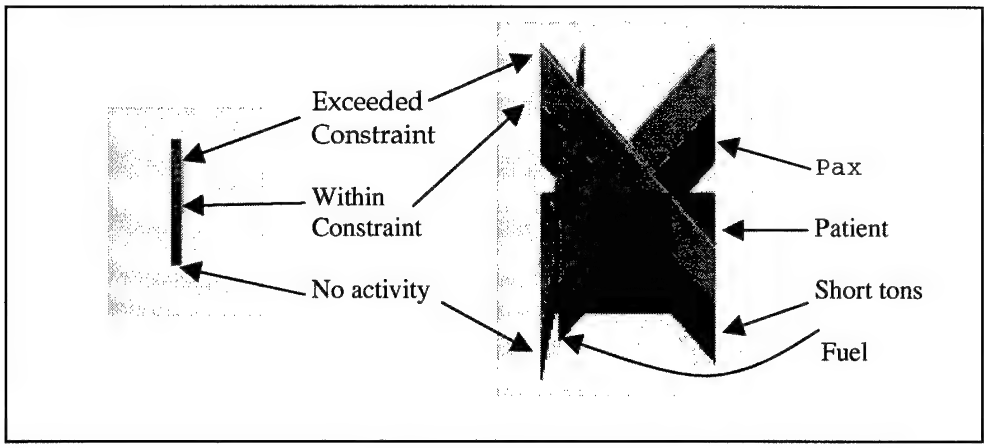
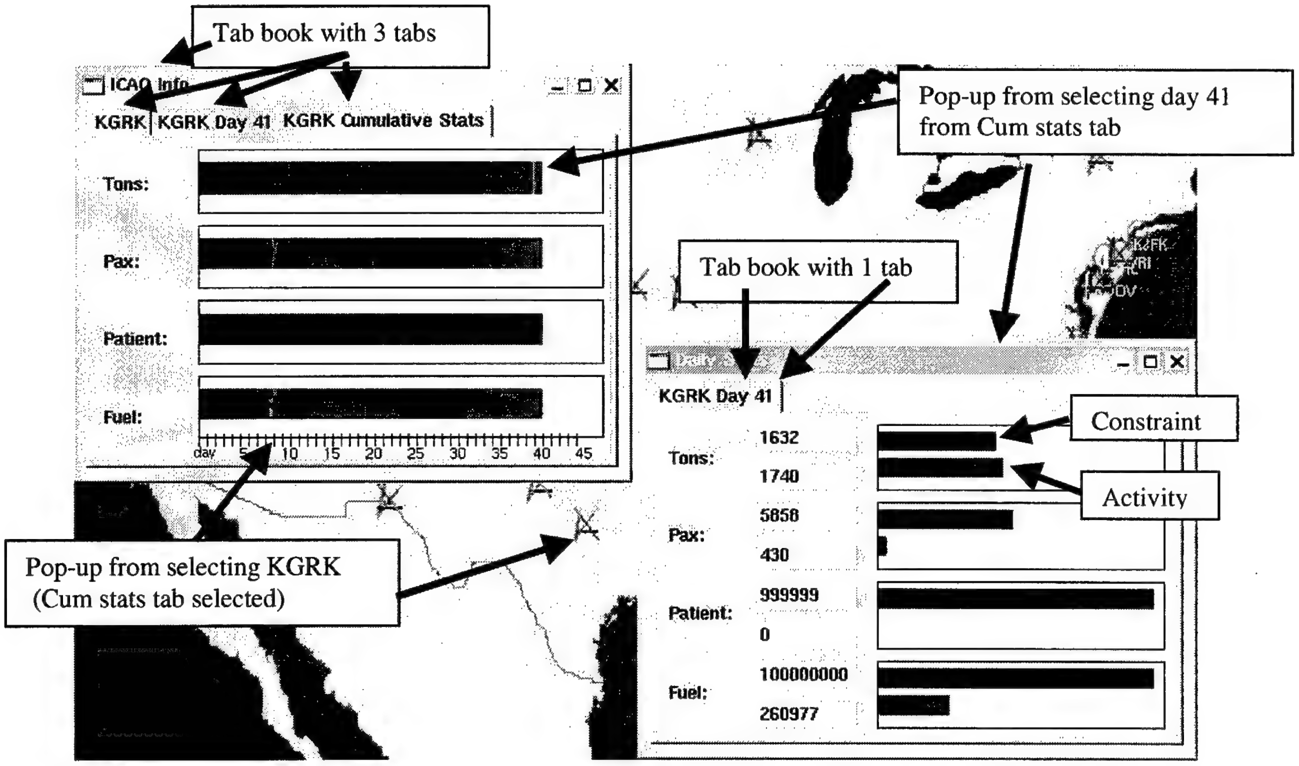


Plate 10: Cumulative Symbols



Tons:	1632	
	1740	
Pax:	5858	
	430	
Patient:	999999	
	0	
Fuel:	100000000	
	260977	

Plate 11: Main Pop-up Window and Day Selection Window

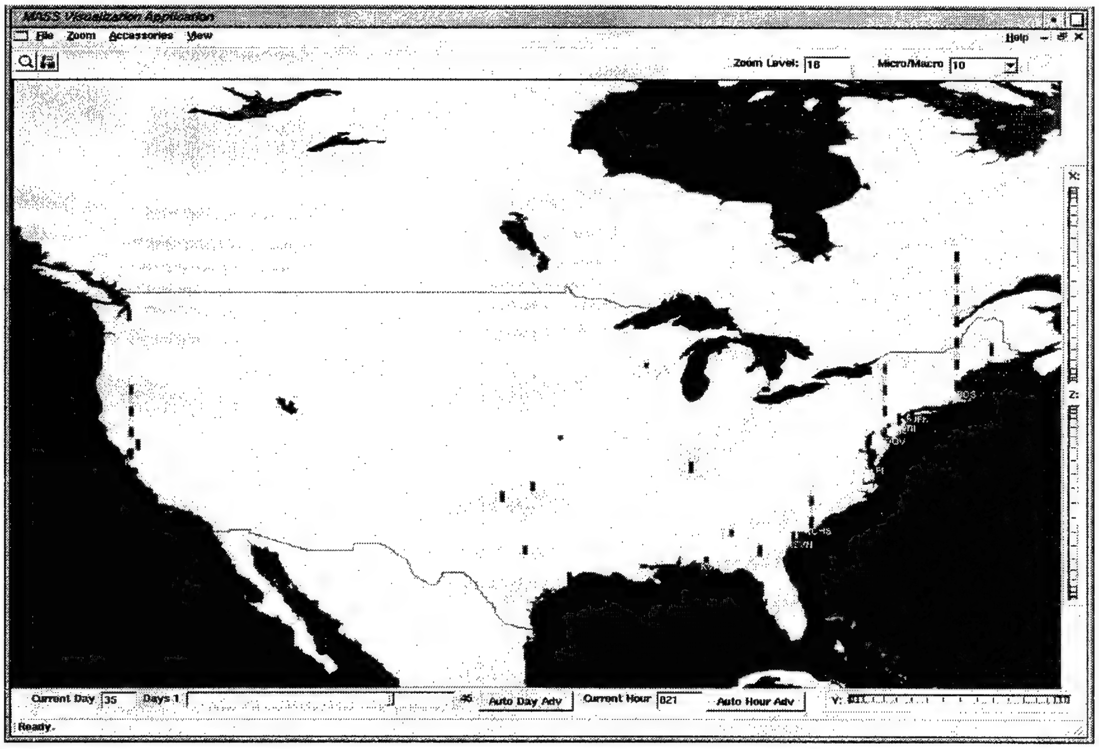


Plate 12: Application #2 Macro View

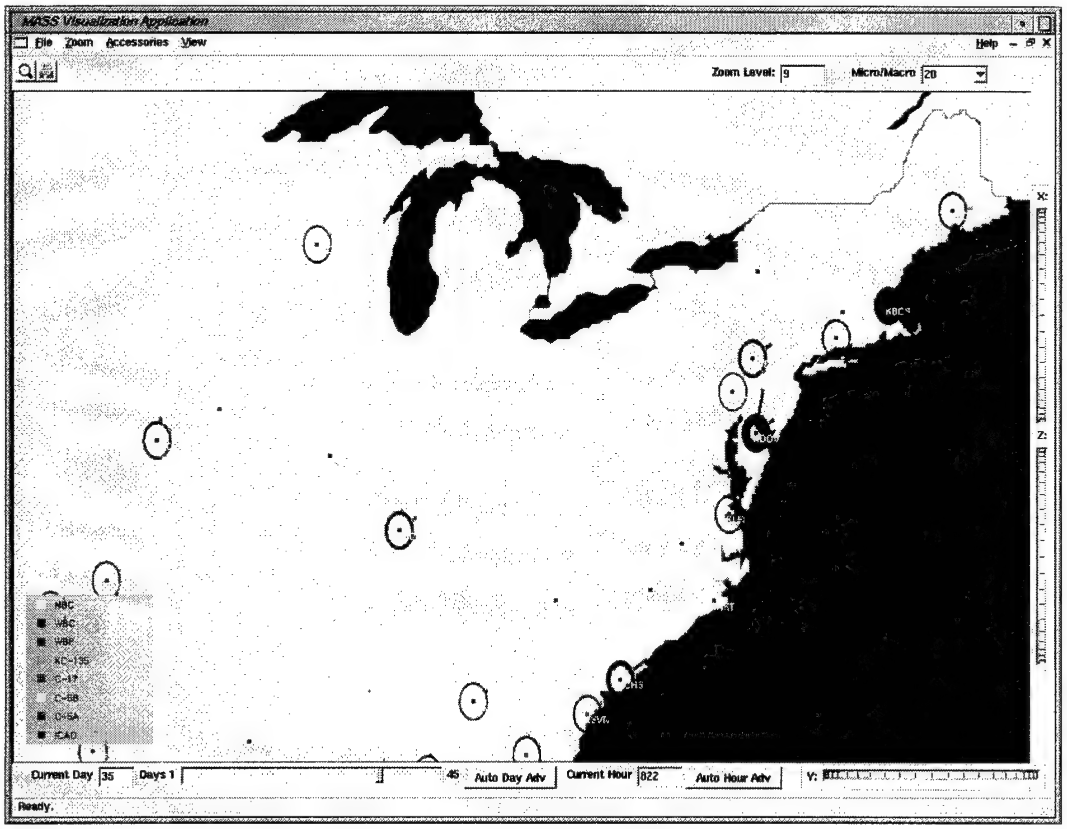


Plate 13: Application #2 Micro View

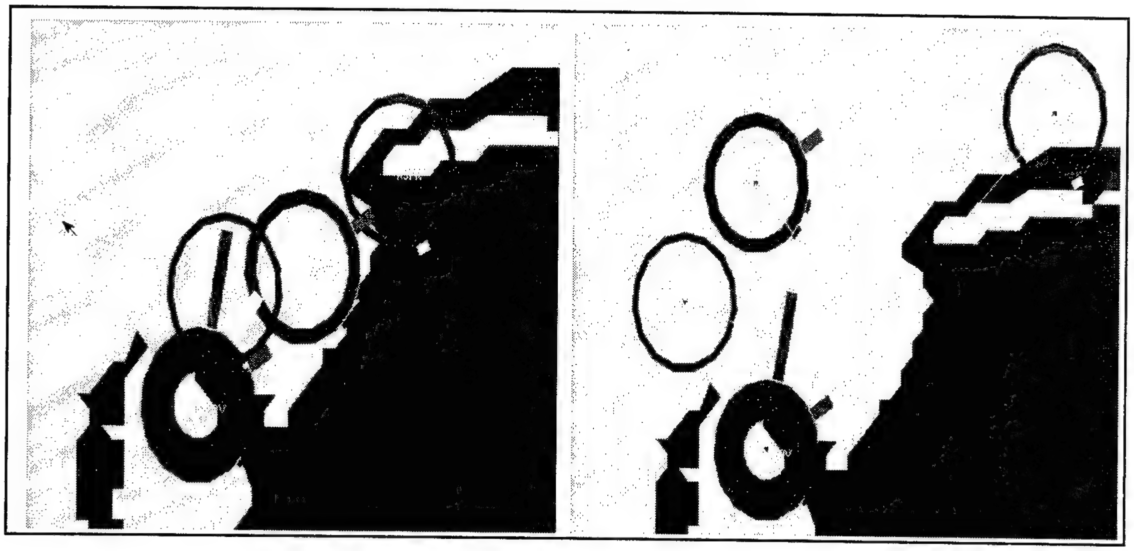
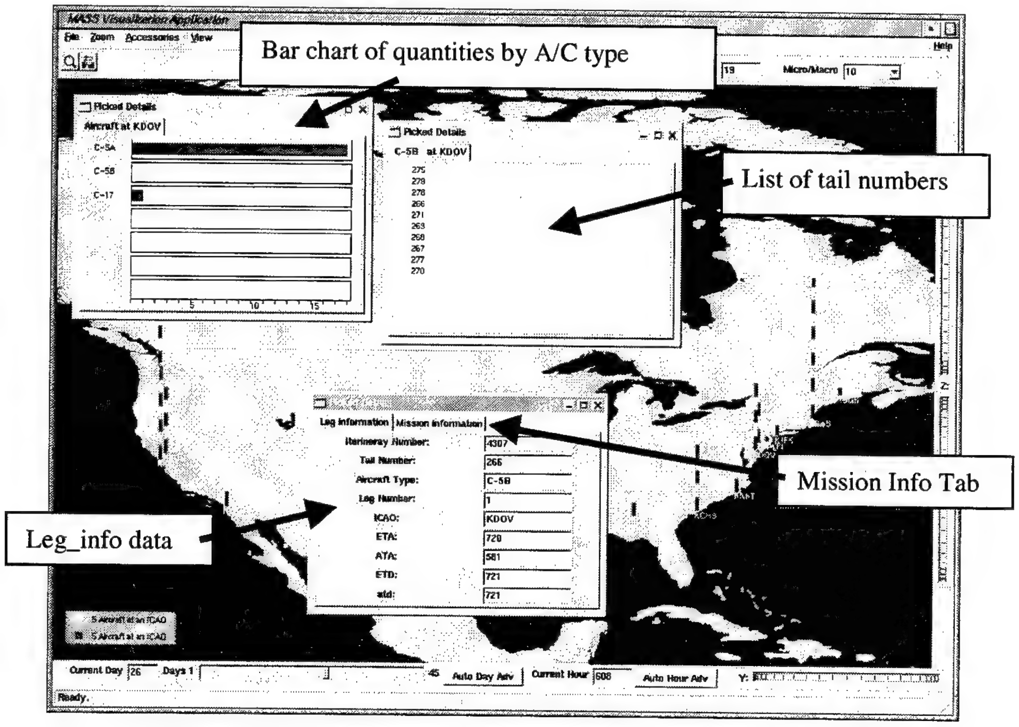


Plate 14: Application #2 Declutter Example



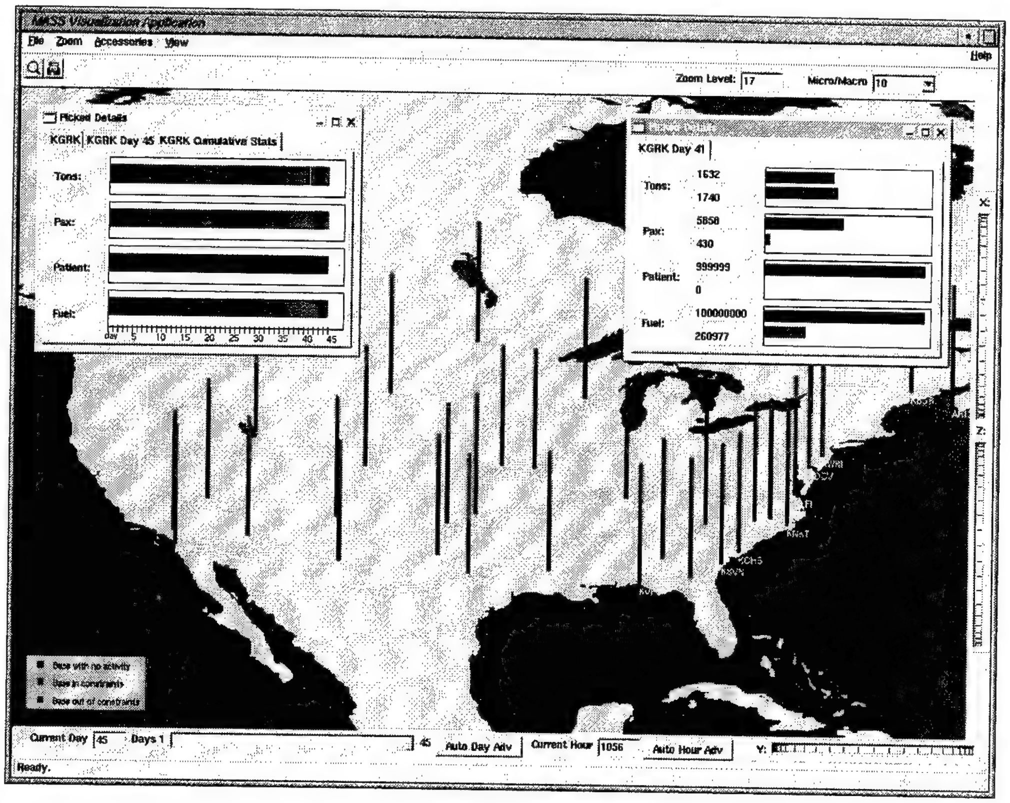
Micro/Macro: 10

C-5B at KDOV: 275, 278, 278, 266, 271, 263, 268, 267, 277, 270

Itinerary Number:	4307
Tail Number:	266
Aircraft Type:	C-5B
Leg Number:	1
ICAO:	KDOV
ETA:	720
ATA:	581
ETD:	721
atd:	721

Current Day 26 Days 1 45 Current Hour 608

Plate 15: Application #2 Drill-down Windows



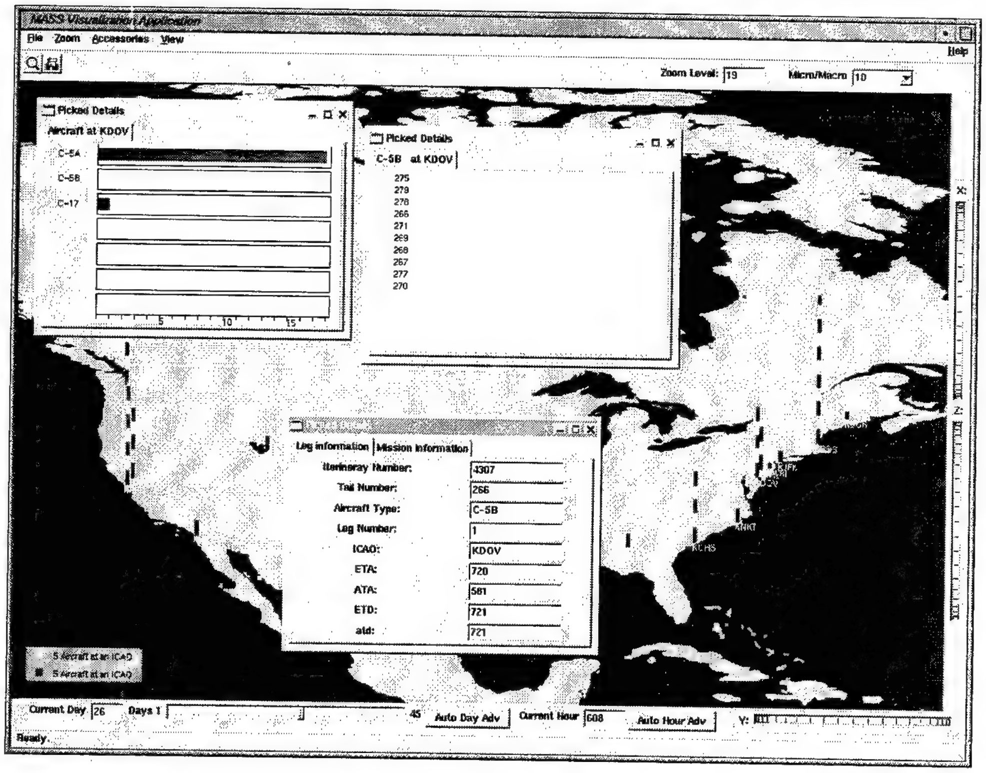
Zoom Level: 17 Micro/Macro 10

KGRK Day 41

Tons:	1632
	1740
Pax:	5858
	430
Patient:	999999
	0
Fuel:	100000000
	260977

Current Day 45 Days 1 … 45 Current Hour 1056

Plate 16: Application #1 Screen Capture



Zoom Level: 19 Micro/Macro 10

C-5B at KDOV: 275, 279, 278, 266, 271, 269, 268, 267, 277, 270

Itterinaray Number:	4307
Tail Number:	266
Aircraft Type:	C-5B
Leg Number:	1
ICAO:	KDOV
ETA:	720
ATA:	581
ETD:	721
atd:	721

Current Day 26 Days 1 … 45 Current Hour 608

Plate 17: Application #2 Screen Capture

Bibliography

1. Air Mobility Command, Scott AFB, Illinois. Airlift Flow Model (Version 10.0) Baseline Document, Appendix A: Input & Output Files, 1999.

2. American Institute. Reference Manual: Hands-On Internetworking in a LAN Environment. American Institute, Inc. 1994.

3. Brown, Judith R. and T. Todd Elvins, 1997. "Euro-American Workshop on Visualization of Information and Data," Computer Graphics, pg. 31 (November 1997).

4. Causse, Sylvain, Frederic Juaneda, and Michel Grave. "Partitioned Objects Sharing for Visualization in Distributed Environments" in Scientific Visualization: Advances and Challenges. pp. 286-305, Rosenblum, 1994.

5. Chuah, Mei C. and Stephen G. Eick. "Information Rich Glyphs for Software Management Data". IEEE Computer Graphics and Applications: July/August 1998. Pp.24-29.

6. CIO-SP-1351D018. Transition and Maintain the Mobility Analysis Support System (MASS) as Department of Defense High-Level Architecture (HLA) Compliant Simulation, Airlift Flow Model Version 10.0 Baseline. Boeing Corporation, Virginia, 16 Oct 1998.

7. Fayad, Mohamed E., Douglas C. Schmidt, and Ralph E. Johnson. "Building Applications Frameworks", Wiley Computer Publishing, New York, 1999.

8. Fox GUI Libraries. http://www.cfdrc.com/fox. 1999.

9. Garlan, David and Mary Shaw, "An Introduction to Software Architecture" in Advances in Software Engineering and Knowledge Engineering. Vincenzo Ambriola and Genoveffa Tortora, World Scientific, New Jersey, 1993.

10. Jerding, Dean F. and John T. Stasko. "The Information Mural: Increasing Information Bandwidth" in Visualizations: Technical Report, October, 1996. GIT-GVU-96-25. Atlanta, GA: Graphics, Visualization, and Usability Center, College of Computing, Georgia Institute of Technology, 1996.

11. Kayloe, Jordan. "Easy-Sim: A Visual Simulation System Software Architecture with an ADA 9X Application Framework. MS Thesis, AFIT/GCS/ENG/94D-11. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1994 (AAL4614).

12. Koutsofios, Eleftherias E., Stephen C. North, and Daniel A Keim. "Visualization Blackboard: Visualizing Large Telecommunication Data Sets," IEEE Computer Graphics and Applications, 16-19, IEEE Press, May/June 1999.

13. Liberty, Jesse. C++ Unleashed. SAMS, Indianapolis, IN., 1998.

14. Lucas, Peter and Steven F. Roth. "Exploring Information with Visage" in IEEE Computer Graphics and Applications, 32-41, IEEE Press, July/August 1997.

15. ----- and Christina C. Bomberg. "Visage: Dynamic Information Exploration". Maya Corp. Excerpt from unpublished article. 1998 http://www.maya.com/Visage/base/papers/mayaDemo.htm.

16. Perry, Greg. "Moving From C to C++: The Ins and Outs of Object-Oriented Programming". SAMS, Indianapolis, IN., 1995.

17. Peterson, John. "The Visually Enabled Enterprise: Management Information Through The Power of Visualization," ObjectFX Corp. Excerpt from unpublished article. http://www.objectfx.com. 1998.

18. Rosenblum, L. and others. Scientific Visualization: Advances and Challenges. Academic Press, New York, 1994.

19. Schroeder, Will, Ken Martin and Bill Lorensen. The Visualization Toolkit: An Object Oriented Approach to 3D Graphics. Chapter 5, Prentice Hall, 1997.

20. Shaw, Mary, and David Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, New Jersey, 1996, pp. 1-3.

21. Sikora, Jim and Phil Coose. "What in the World is ADS?" PHALANX: The Bulletin of Military Operations Research. Vol. 28, No. 2, June 1995.

22. Woo, Mason, Jackie Neider and Tom Davis. OpenGL Programming Guide. 2nd Ed. Addison-Wesley, California, 1998.

Vita

Captain Stuart H. Kurkowski was born 3 October 1968 in Rochester, Minnesota. He graduated from Terry Parker High School in Jacksonville, Florida in June 1987. He completed his undergraduate studies at The United States Air Force Academy in Colorado Springs, Colorado where he graduated with a Bachelor of Science degree in Computer Science and was commissioned in May 1991.

His first assignment was at Reese AFB as a student in Undergraduate Pilot Training in October 1991. In May 1992, he was assigned to the 1912th Computer Systems Group, Langley AFB, Virginia where he served as a computer programmer and software engineer. While in Virginia, he entered graduate studies at Troy State University in Hampton, Virginia where he graduated with a Masters of Science in Information Management in December of 1994. In June of 1995 he was assigned to the Defense Information Systems Agency in the Global Command and Control Systems office. In August 1998, he entered into the Computer Science Masters program, School of Engineering, Air Force Institute of Technology. Upon graduation, he will be assigned to the AFOTEC Detachment 4 Peterson AFB, Colorado.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** March 2000	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE AN INFORMATION VISUALIZATION SOLUTION FOR THE ANALYSIS OF THE AFM SIMULATION DATA	**5. FUNDING NUMBERS**
6. AUTHOR(S) Stuart H. Kurkowski, Captain, USAF	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology, Graduate School of Engineering and Management (AFIT/EN), 2950 P Street, Building 640, WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GCS/ENG/00M-11
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ AMC/XPY, Attn: Maj. Robert T. Brigantic, 402 Scott Drive, Unit 3L3, SCOTT AFB, IL DSN: 576-8713	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

Lt. Col. Timothy Jacobs, ENG

12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

ABSTRACT (*Maximum 200 Words*)

This thesis looks at developing a robust information visualization architecture that integrates data processing, visualization, and user interaction, and supports reuse and component-based functions. This research develops a component-based 3-D visualization system for the AFM data. A domain-independent application framework is developed to support the component-based system design. This research also develops data reading objects, integrated data structures, and visual components as well as drill-down and user-interface components to produce an end-to-end visualization application for several aspects of the AFM data.

The results of this research show that an application framework can support information visualization applications. The use of a stable underlying framework architecture provides high levels of design and code reuse for future component development. The component-based functionality frees future development to concentrate on visualizing data and not the systemic concerns handled by the framework. This enables AFM and others to get a better return on investment for future work. The representative applications completed in this research already provide AMC with unprecedented insight into the AFM data.

14. SUBJECT TERMS Information Visualization, Software Architecture, Simulation Analysis, Computer Graphics, Visualization Architectures	**15. NUMBER OF PAGES** 104
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL